Inferencia de tipos seguros en un lenguaje funcional con destrucción explícita de memoria

Proyecto de Fin de Máster

Autor: **Manuel Montenegro Montes**
Profesores directores: **Ricardo Peña Marí y Clara María Segura Díaz**
Facultad de Informática
Universidad Complutense de Madrid
Septiembre 2007

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mis directores de proyecto, Clara Segura y Ricardo Peña por su inestimable ayuda y dedicación.

Agracezco a Estela (*Copyprina*) y a Gerardo (*D.P.*) por su demostrada paciencia en el día de entrega del presente trabajo. También a Alfredo, Sergio, Raúl, Roberto y Juanan por su alegre compañía (en forma de icono ) durante la redacción de este documento: es una lástima que en los últimos meses antes de su entrega tuviese que desinstalar tan malintencionado software de mi sistema.

Índice general

Capítulo 1

Introducción

En este trabajo se describirá con detalle el diseño e implementación de un algoritmo de inferencia para un lenguaje funcional con manejo de regiones y destrucción explícita de memoria, llamado SAFE. Este algoritmo permite trabajar de modo seguro con estructuras de datos, ya que a partir del código fuente de un programa escrito en SAFE garantiza que durante la ejecución del mismo no se producirán accesos a zonas de la memoria ya liberadas.

Antes de embarcarse en los detalles resulta necesario ofrecer una visión global del contexto en el que el lenguaje SAFE se encuentra situado, así como una motivación que justifique la necesidad de un lenguaje que proporcione un enfoque semiexplícito de manejo de memoria. Esto será el objetivo de este capítulo.

1.1. Motivación

La programación funcional está contenida dentro de un modelo de programación conocido como *declarativo*. En dicho modelo los programas están centrados es la especificación de los cómputos que se realizan, evitando detalles sobre el modo en el que esto ocurre. En este sentido los lenguajes funcionales están concebidos para ofrecer un mayor nivel de abstracción que los imperativos. Dentro del paradigma funcional los programas están basados en el concepto matemático de *función*, definida como un conjunto de *ecuaciones*. La programación funcional tiene como modelo de cómputo subyacente el *λ-cálculo*, bajo el cual se trabaja sobre expresiones, reduciéndolas sucesivamente hasta obtener un resultado.

Los programas escritos en lenguajes funcionales gozan de ciertas propiedades que resultan deseables desde el punto de vista del programador. Una de ellas es la *transparencia referencial*: lo que define una función es la correspondencia que realiza entre los elementos de su dominio y su rango. Dicho de otra forma, el resultado de evaluar una función depende exclusivamente de sus parámetros de entrada. Esto implica que una expresión se evaluará siempre al mismo valor, independientemente del contexto en el que se encuentre. Una consecuencia de esto es la ausencia del concepto de *estado* en programación funcional, lo cual facilita la tarea de razonar sobre la corrección

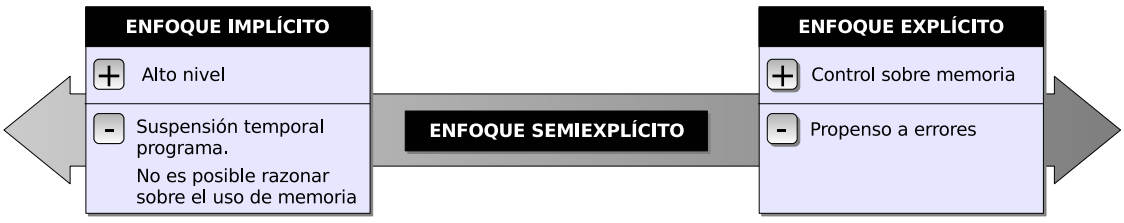

Figura 1.1: Manejo de memoria en lenguajes funcionales

de un programa. En este contexto, los programas funcionales se prestan con relativa facilidad a su *análisis estático*, mediante el cual a partir del texto del programa se infieren propiedades del mismo. Uno de los análisis más extendidos en los lenguajes funcionales modernos es el *análisis de tipos*: las expresiones se clasifican según el tipo de valores a los que pueden ser reducidas. Esto garantizará que no se producirán errores de ejecución en el programa por incompatibilidades de tipo.

Dentro de los detalles que los lenguajes funcionales abstraen, puede encontrarse el *manejo de memoria* como un caso particular: En la mayoría de lenguajes funcionales el sistema de ejecución reserva memoria a medida que se necesite, mientras haya espacio disponible. En caso de que se agote, un sistema de recolección de basura detecta aquellas partes de la memoria no son accesibles desde el estado actual de ejecución del programa y, por tanto, pueden reutilizarse.

Sin embargo, en determinadas situaciones (aplicaciones críticas en seguridad, sistemas de tiempo real) resultan inaceptables la suspensión temporal del programa para ejecutar el recolector de basura y la parada repentina del programa en caso de que se agote la memoria disponible.

Por otra parte, muchos lenguajes imperativos (como por ejemplo C++) permiten que el programador asuma (por ejemplo, mediante operadores `new` y `delete`) la reserva y la liberación de memoria. No obstante, este tipo de mecanismo es fuente de numerosos errores: punteros descolgados, memoria no liberada en la terminación del programa, compartición indeseada entre estructuras, etc.

1.2. Gestión de memoria en programas funcionales

En relación al manejo de memoria en lenguajes funcionales podemos encontrar varios enfoques: en [HJ03] se propone un análisis en tiempo de compilación para obtener cotas superiores lineales sobre el consumo de memoria producido por un programa. Los programas son escritos en un lenguaje funcional con una construcción especial llamada match que libera celdas de memoria. En [TT97, TBE+06] se ofrece una disciplina de gestión de memoria basada en regiones (esto es, zonas disjuntas de memoria donde los valores son almacenados). Los lugares en el flujo de programa donde se reservan y liberan regiones son inferidos automáticamente. En relación al manejo de memoria basado en regiones se ofrece en [HP99a] un método para comprobar cotas superiores

de espacio. No obstante, en ninguno de estos trabajos se ofrece un análisis que garantice que la destrucción de estructuras de datos se realiza de forma segura, detectando así en tiempo de compilación los problemas provenientes del manejo explícito de la memoria comentados anteriormente.

En el DSIC se ha desarrollado un enfoque semiexplícito de manejo de la memoria, mediante la definición de un lenguaje funcional llamado SAFE, cuya descripción puede encontrarse en el Capítulo 2. En dicho lenguaje el programador especifica cierta información sobre el uso de las estructuras de datos. En particular puede indicar que una determinada estructura de datos no va a utilizarse más y que, por consiguiente, la región de memoria en la que se encuentra puede ser reutilizada. Los beneficios de este enfoque son:

1. Un sistema de tipos garantiza que el uso de estas facilidades de destrucción de estructuras pueden realizarse de forma segura. En particular se evita el acceso a estructuras de datos ya liberadas.

2. Es posible prescindir de un sistema de recolección de basura. Esto facilita la posibilidad de razonar sobre la cantidad de memoria que consume un programa.

1.3. Objetivos

El trabajo aquí presentado tiene como objetivo el **diseño e implementación del algoritmo de inferencia de tipos seguros para programas escritos en el lenguaje SAFE**. No obstante, para poder realizar dicha inferencia son necesarios algunos análisis previos, concretamente:

- **Inferencia de tipos Hindley-Milner:** Se encarga de asignar a cada expresión de un programa su tipo correspondiente. Éste es el algoritmo comunmente utilizado en lenguajes funcionales con disciplina de tipos, como Haskell. La parte fundamental del mismo procede en dos fases: generación de un sistema de ecuaciones (o *unificaciones*) entre tipos y resolución de dicho sistema. En el caso concreto del lenguaje SAFE se han de incorporar además en dicho algoritmo ciertas particularidades propias provenientes en gran medida del hecho de utilizar un enfoque basado en regiones como sistema de manejo de memoria.

- **Análisis de compartición:** Tiene como objetivo aproximar las relaciones de compartición que pueden producirse entre las estructuras de datos durante la ejecución de un programa.

Con el resultado de estos dos análisis puede abordarse el análisis de **tipos seguros** (también llamados *tipos SAFE*), que extienden a los tipos Hindley-Milner incluyendo información adjunta sobre el uso previsto de las estructuras de datos. En concreto especifican si dichas estructuras resultan directamente destruidas durante una llamada a una función, o apuntan a una celda de memoria que ya ha sido liberada. Durante la

inferencia de estos tipos se tendrá en cuenta esta nueva información para garantizar que las destrucciones no se realizan de forma insegura.

Los análisis y algoritmos explicados en este trabajo han sido implementados por el autor del mismo en el marco de la parte frontal de un compilador para el lenguaje SAFE. En el presente trabajo se expondrán algunos detalles relativos a esta implementación.

Las aportaciones originales del autor de este trabajo destinadas a satisfacer estos objetivos planteados se muestran a continuación.

1. Aspectos de implementación de la inferencia de tipos Hindley-Milner, incluyendo la extensión ya mencionada del algoritmo de inferencia Hindley-Milner original.

2. Implementación del análisis de compartición de estructuras. Este análisis resulta imprescindible para conocer si la liberación de memoria se realiza de forma segura. Se estudiarán las estructuras de datos utilizadas que permiten un análisis eficiente.

3. Coautor del diseño del algoritmo de inferencia de tipos seguros y autor de la implementación del mismo. Como ya se ha explicado, el análisis realizado por este algoritmo permite garantizar si un programa no realiza operaciones inseguras, como por ejemplo, el acceso a memoria ya liberada.

4. Demostración de corrección del algoritmo de inferencia de tipos seguros con respecto al sistema de tipos SAFE.

El trabajo realizado hasta el momento ha dado lugar a dos publicaciones: en [PSM07] se describe el análisis de compartición entre estructuras de datos y en [MPS07] se detalla el algoritmo de inferencia de tipos seguros.

1.4. Plan de trabajo

Este trabajo describe algunas de las etapas que conforman el compilador de SAFE. El orden en que estas etapas serán explicadas en este trabajo se corresponde con el orden en el que se suceden durante la compilación.

- En el Capítulo 2 se introducirá el lenguaje SAFE. Antes de definir su sintaxis y su semántica, se explicarán los conceptos y decisiones principales que motivaron su diseño.

- El Capítulo 3 describirá el diseño e implementación de la fase de inferencia de tipos Hindley-Milner aplicado al lenguaje SAFE. Comprenderá tanto el algoritmo de inferencia general como las fases específicas a este lenguaje

- Por su parte, el Capítulo 4 tratará el diseño e implementación del análisis que aproxima las relaciones de compartición entre variables. Se definirán en primer lugar los distintos tipos de relaciones de compartición que pueden producirse entre estructuras de datos. A continuación se describirá la forma en la que se infieren dichas relaciones a partir de las expresiones que conforman un programa.
- La información obtenida de la inferencia de tipos Hindley-Milner y el análisis de compartición sirven como base para el algoritmo de inferencia de tipos SAFE, que será descrito en el Capítulo 5. Antes de describir el propio algoritmo, se introducirá el sistema de tipos que garantiza que no se crean punteros descolgados en memoria. Por otra parte, se detallarán al final del capítulo los aspectos de implementación más relevantes en este algoritmo.
- El Capítulo 6 se encargará de establecer la relación entre el algoritmo de inferencia de tipos seguros y el sistema de tipos, ambos presentados en el capítulo anterior. En primer lugar se ofrecerá la demostración para determinadas propiedades del algoritmo de inferencia de tipos SAFE. Estas propiedades serán utilizadas para demostrar la corrección de dicho algoritmo de inferencia con respecto al sistema de tipos SAFE.
- En último lugar, el Capítulo 7 expondrá las conclusiones obtenidas y el trabajo futuro a realizar en el marco del lenguaje SAFE.

Capítulo 2

Descripción del lenguaje SAFE

Este capítulo ofrecerá una breve introducción al lenguaje SAFE. En primer lugar se describirá el contexto del proyecto SAFE y el área de investigación en el que se encuentra situado. Posteriormente se definirá el lenguaje *Core-Safe* y su versión edulcorada *Full-Safe*. Se pondrá especial atención en el primero, ya que es el lenguaje en el que se realizan la mayoría de los análisis explicados en este trabajo.

Una vez expuestas la sintaxis y semántica del lenguaje se mostrarán varios ejemplos que ilustran las funciones de liberación explícita de memoria ofrecidas por el mismo. Por último se describirá brevemente el proceso de compilación de un programa SAFE.

2.1. Marco del proyecto

En el Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid, el grupo de Programación Funcional viene participando en un proyecto (TIN2004-07943-C04) coordinado con las Universidades de Castilla-La Mancha (UCLM), Politécnica de Valencia (UPV) y de Málaga (UMA). Dicho proyecyo recibe el nombre de SELF (*Software Engineering and Lightweight Formalisms*). En el seno del mismo, el grupo de la UCM trabaja en el campo de investigación del Código con Demostración Asociada (*Proof Carrying Code* o PCC). Con el fin de ofrecer una visión global del contexto en el que se encuentra situado el lenguaje SAFE, se ofrece a continuación una breve descripción de las líneas de investigación en las que el proyecto SELF está situado.

2.1.1. SELF (*Software Engineering and Lightweight Formalisms*)

Uno de los problemas actuales dentro del desarrollo de tecnología software es la dificultad cada vez mayor para poder garantizar la fiabilidad de los sistemas desarrollados. El proyecto tiene como objetivo el desarrollo de métodos, herramientas y técnicas necesarias para permitir la creación de software de calidad, estando enfocado en la aplicación práctica a los procesos llevados a cabo por compañías de software.

La idea básica del proyecto es el uso de métodos formales ágiles aplicados a Ingeniería del Software, es decir, la aplicación parcial de formalismos a distintos niveles de desarrollo: lenguaje, modelado, análisis y composición. De este modo se trata de mejorar la aplicación real de métodos formales en ciertas fases del ciclo de vida del software. Las actividades realizadas en este contexto pueden enmarcarse en una de estas áreas de investigación:

- **Descripción semántica de componentes software:** Análisis de las distintas posibilidades ofrecidas por los lenguajes de especificación y lenguajes funcionales para describir sistemas software. Desarrollo de técnicas para analizar la terminación de especificaciones, su eficiencia y posibles optimizaciones a aplicar.
- **Verificación y depuración del software:** Mejora de las técnicas de *model checking* y código con demostración asociada (PCC), que proporcionan una validación automática de los sistemas software. Dentro de este campo, el proyecto está enfocado a la aplicación de técnicas PCC en entornos de programación declarativa.
- **Optimización de programas multiparadigma:** Los lenguajes declarativos multiparadigma (lógicos, funcionales, concurrentes, con restricciones, etc.) ofrecen un alto poder de expresividad. El objetivo es la mejora de la eficiencia de estos lenguajes.

2.1.2. Código con Demostración Asociada (PCC)

El lenguaje SAFE se encuentra en el área de investigación del Código con Demostración Asociada. En este marco, un consumidor de código verifica que el código proporcionado por un productor de código no fiable satisface un conjunto de reglas, que constituyen su *política de seguridad*. La verificación se realiza mediante la comprobación de una *demostración formal* que el productor adjunta al código que genera.

Dentro del ámbito del lenguaje, la política de seguridad que conforma el principal objetivo del proyecto es la certificación del *manejo seguro de la memoria*, que implica:

1. La garantía de que no se realiza acceso a punteros descolgados en memoria.
2. La existencia de una cota superior en la cantidad de memoria necesaria para ejecutar un programa. Dicha demostración se incluye en el programa.

Estos dos elementos son enfocados desde la perspectiva del análisis de programas. Dichos análisis se basan tanto en las técnicas de interpretación abstracta como en sistemas de tipos y efectos.

2.1.3. SAFE

El lenguaje fuente sobre el que se realizan los distintos análisis comentados anteriormente recibe el nombre de SAFE. Se trata de un lenguaje funcional impaciente

de primer orden con facilidades para la destrucción explícita y copia de estructuras de datos. Por otro lado proporciona *regiones* (es decir, partes disjuntas de la memoria) donde el programador puede almacenar estructuras de datos.

Este trabajo se centrará en dos análisis: compartición entre estructuras de datos e inferencia de tipos seguros. Este último proporciona la base para la generación del certificado de una de las reglas de la política de seguridad particular de este lenguaje: la imposibilidad de acceso a punteros descolgados en memoria.

2.2. El lenguaje SAFE

El objetivo principal de esta sección es presentar la sintaxis y semántica del lenguaje funcional SAFE. Antes de eso es necesario introducir los conceptos y decisiones de diseño que subyacen en el mismo. En [PS04] puede encontrarse una explicación más detallada de las definiciones aquí enunciadas y una motivación de las distintas decisiones tomadas. Existen dos versiones del lenguaje SAFE:

- *Full-Safe*: Lenguaje en el cual se escriben los programas. Posee una sintaxis similar a la de Haskell, incorporando el uso de variables de región, ajuste de patrones destructivo y operaciones de copia y reutilización.
- *Core-Safe*: Lenguaje núcleo sobre el que se definen los distintos análisis. Se trata de un subconjunto de *Full-Safe*.

Dentro del proceso de compilación, descrito con detalle en la Sección 2.4, existe una fase de traducción de programas *Full-Safe* a programas *Core-Safe*. El análisis de compartición y la inferencia de tipos seguros se definen sobre este último y, por tanto, se realizan tras esta fase de transformación.

EJEMPLO 1. *A continuación se muestra la definición en lenguaje Full-Safe de una función que calcula la longitud de una lista:*

$$\begin{array}{l} \mathit{length}\ [\,] = 0 \\ \mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs \end{array}$$

Puede observarse que la definición es idéntica a la que podría encontrarse en Haskell, ya que no se hace uso de ninguna de las funciones de manejo explícito de memoria proporcionadas por SAFE. No obstante se incluye aquí este ejemplo para ser comparado con su equivalente Core-Safe, que se muestra a continuación:

$$\begin{array}{l} \mathit{length}\ ys = \textbf{case}\ ys\ \textbf{of} \\ \qquad\qquad [\,] \rightarrow 0 \\ \qquad\qquad (x : xs) \rightarrow \textbf{let}\ n = \mathit{length}\ xs\ \textbf{in}\ 1 + n \end{array}$$

Pueden encontrarse diferencias significativas. El ajuste de patrones, que en la definición Full-Safe quedaba implícito mediante el uso de distintas ecuaciones, ahora se indica explícitamente mediante la construcción **case**. *Por otro lado, en Core-Safe se obliga a que los parámetros*

de una función sean literales o variables. Por tanto, la llamada recursiva a la función length ha tenido que ser separada de la aplicación del operador (+) mediante la construcción **let**.

Otra diferencia entre Full-Safe y Core-Safe es la no existencia de constructoras y funciones infijas en este último. Además existe un renombramiento de variables previo al proceso de transformación, por lo que el código Core-Safe generado realmente por el compilador se asemejaría más a lo que se muestra a continuación:

$$\begin{array}{l} \mathit{length}\ \mathit{30\_x} = \textbf{case}\ \mathit{30\_x}\ \textbf{of} \\ \qquad\qquad\qquad [\,] \rightarrow 0 \\ \qquad\qquad\qquad (:)\ \mathit{31\_x}\ \mathit{32\_x} \rightarrow \textbf{let}\ \mathit{33\_x} = \mathit{length}\ \mathit{32\_x}\ \textbf{in}\ (+)\ 1\ \mathit{33\_x} \end{array}$$

No obstante, por motivos de claridad, en los ejemplos Core-Safe mostrados en este capítulo y posteriores no se tendrá en cuenta el renombramiento y se permitirán funciones/constructoras infijas.

Antes de comentar los detalles del lenguaje se presentarán los conceptos fundamentales en la definición del mismo: regiones, celdas y estructuras de datos.

DEFINICIÓN 1. *Una **región** es un área contigua de memoria en el montón donde se puede construir, leer o destruir estructuras de datos. Tiene un tamaño conocido en tiempo de ejecución y puede ser reservada y liberada completamente en tiempo constante.*

DEFINICIÓN 2. *Una **celda** es un pequeño espacio de memoria lo suficientemente grande como para almacenar un constructor de datos. En términos de implementación, una celda contiene la marca (o puntero al código) del constructor, y una representación de las variables libres a las que se aplica. Los parámetros del constructor pueden ser valores básicos, o bien punteros a valores no básicos.*

Por el término "lo suficientemente grande" se entiende que una celda liberada puede ser reutilizada directamente por el sistema. Una implementación ingenua de ello consiste en incluir en una celda suficiente espacio como para poder almacenar el constructor más grande. Un enfoque más eficiente puede consistir en tener un número fijo de tamaños de celda, todos ellos múltiplos del tamaño más pequeño. La idea fundamental consiste en poder reutilizar una celda en un tiempo constante.

DEFINICIÓN 3. *Una **estructura de datos** (ED) es un conjunto de celdas que se obtiene empezando por una celda considerada como la raíz, y hallando el cierre transitivo de la relación $C_1 \rightarrow C_2$, donde C_1 y C_2 son celdas del mismo tipo T y en C_1 hay un puntero a C_2.*

Por ejemplo, en una lista de listas (tipo `[[a]]`), las celdas contenidas en la estructura cons-nil de la lista *más externa* forman una ED, en la que no están incluidas las listas internas. Cada una de estas últimas constituye una ED propia.

Durante el diseño del lenguaje se tomaron las siguientes decisiones:

1. *Una ED reside completamente en una región.* Esta decisión impone una restricción en las constructoras de datos: los hijos recursivos de una celda han de pertenecer a la misma región que ésta.

2. *Una ED puede ser parte de otra, o dos EDs pueden compartir una tercera.*

3. *Los valores básicos (enteros y booleanos) no reservan celdas en ninguna región. Se encuentran dentro de las celdas de una ED, o en la pila.*

4. *Una función que recibe n parámetros de entrada puede acceder, como mucho, a $n + 2$ regiones.*

   - Puede acceder a las EDs de sus parámetros, posiblemente residiendo cada uno en una región distinta.
   - Toda función que devuelva una ED como resultado debe construirla en la **región de salida**. Como máximo hay una región de salida por cada función. La expresión que realice la llamada a dicha función es la responsable de suministrar el identificador de región correspondiente. La función recibe la región de salida como un parámetro adicional.
   - Toda función posee una **región de trabajo** opcional, denominada mediante el identificador *self*, donde pueden construirse EDs intermedias. La región de trabajo tiene la misma duración que la llamada a la función: Se reserva al inicio de la llamada a la función y se libera cuando su ejecución termina.

5. *Si un parámetro de entrada a una función es una ED, puede ser destruido por la misma.* En ese caso se dice que el parámetro está **condenado**, porque esta condición depende de la definición de la función; no de su uso. Por 'destrucción' se entiende que el invocador de la función puede asumir con seguridad que las celdas ocupadas por la ED condenada son recuperadas por el sistema de ejecución.

6. *Toda función tiene las siguientes posibilidades sobre sus EDs y regiones accesibles:*

   - *Una función sólo puede leer una ED que aparece como parámetro de sólo lectura.*
   - *Una función puede leer (antes de destruir) y debe destruir una ED que es un parámetro condenado.*
   - *Una función puede construir, leer y destruir EDs en su región de salida y en su región de trabajo.*

2.2.1. Sintaxis *Core-Safe*

La sintaxis para el lenguaje funcional *Core-Safe* se muestra en la Figura 2.1. Se trata de un lenguaje de primer orden donde se impone la compartición mediante el uso de variables en las aplicaciones de funciones y constructoras.

Un programa en *Core-Safe*, *prog*, es una secuencia de definiciones de funciones polimórficas recursivas simples seguidas por una expresión principal *expr*, cuyo valor es el resultado del programa. Toda definición de función sólo puede contener llamadas a funciones previamente definidas y la expresión principal puede llamar a cualquiera de ellas. Si una función construye una ED entonces tendrá un parámetro adicional *r*

prog	$\rightarrow$	$dec_1; \ldots; dec_n; expr$	
dec	$\rightarrow$	$f\ \overline{x_i}^n\ @r = expr$	{función polimórfica, recursiva simple}
		$\mid f\ \overline{x_i}^n = expr$	
expr	$\rightarrow$	a	{átomo: literal c o variable x}
		$\mid x@r$	{copia}
		$\mid x!$	{reutilización}
		$\mid (f\ \overline{a_i}^n)@r$	{aplicación de función que construye ED}
		$\mid (f\ \overline{a_i}^n)$	{aplicación de función}
		$\mid (C\ \overline{a_i}^n)@r$	{aplicación de constructor}
		$\mid$ **let** $x_1 = expr_1$ **in** $expr$	{no recursivo, monomórfico}
		$\mid$ **case** x **of** $\overline{alt_i}^n$	{case de sólo lectura}
		$\mid$ **case!** x **of** $\overline{alt_i}^n$	{case destructivo}
alt	$\rightarrow$	$C\ \overline{x_i}^n \rightarrow expr$	

Figura 2.1: Sintaxis del lenguaje *Core-Safe*

que representa la región de salida donde la ED resultante será construida. Las únicas variables de región que pueden aparecer en el lado derecho de una definición son la región de salida r y la región de trabajo *self*.

Por otro lado, se permiten declaraciones de tipos de datos algebraicos polimórficos. Estos tipos se supondrán definidos aparte mediante declaraciones **data**. Existe un tipo de datos predefinido, $[a]$, que representa a las listas de elementos de tipo a. Este tipo de datos incluye los constructores $[\,]$ y (:).

Con respecto a las expresiones, se incluyen las construcciones habituales en lenguajes funcionales: variables, literales, aplicaciones de función y constructor, construcciones **let** y **case**. No obstante, existen expresiones adicionales que serán descritas a continuación:

- Si x apunta a una ED, la expresión $x@r$ representa una **copia** de la ED apuntada por x, suponiendo que x no resida en r. Por ejemplo, si x es una lista contenida en $r' \neq r$, entonces $x@r$ es una lista nueva con la misma estructura que x, pero compartiendo sus elementos.

- La expresión $x!$ indica la **reutilización** de la ED destruíble a la que x apunta. La reutilización resulta útil cuando no se quiere destruir un parámetro condenado completamente, sino reutilizar parte del mismo. Otra posibilidad (menos eficiente) sería realizar una copia previa de x, permitiendo posteriormente destruir el original.

- La expresión **case**! x **of** $\cdots$ indica que el constructor más externo de la ED apuntada por x es liberado tras el ajuste de patrones, de modo que x ya no es accesible. Las subestructuras recursivas de x podrán ser destruidas en el código posterior mediante otro **case**! o reutilizadas mediante el símbolo !. Una variable condena-

$$\Delta, k : c \Downarrow \Delta, k : c \quad [Lit]$$

$$\Delta, k : C\,\overline{a_i}^n@j \Downarrow \Delta, k : C\,\overline{a_i}^n@j \quad [Cons]$$

$$\Delta[p \mapsto w], k : p \Downarrow \Delta, k : w \quad [Var_1]$$

$$\frac{j \leq k \quad l \neq j \quad (\Theta, C\,\overline{a'_i}^n) = copy(\Delta, j, C\,\overline{a_i}^n)}{\Delta[p \mapsto (l, C\,\overline{a_i}^n)], k : p@j \Downarrow \Theta, k : C\,\overline{a'_i}^n@j} \; [Var_2]$$

$$\Delta \cup [p \mapsto w], k : p! \Downarrow \Delta, k : w \quad [Var_3]$$

$$\frac{\Sigma \vdash f\,\overline{x_i}^n = e \quad \Delta, k+1 : e[\overline{a_i/x_i}^n, k+1/self] \Downarrow \Theta, k'+1 : v}{\Delta, k : f\,\overline{a_i}^n \Downarrow \Theta\,|_{k'}, k' : v} \; [App_1]$$

$$\frac{\Sigma \vdash f\,\overline{x_i}^n\,@r = e \quad \Delta, k+1 : e[\overline{a_i/x_i}^n, k+1/self, j/r] \Downarrow \Theta, k'+1 : v}{\Delta, k : f\,\overline{a_i}^n@j \Downarrow \Theta\,|_{k'}, k' : v} \; [App_2]$$

$$\frac{\Delta, k : e_1 \Downarrow \Theta, k' : c \quad \Theta, k' : e[c/x_1] \Downarrow \Psi, k'' : v}{\Delta, k : \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e \Downarrow \Psi, k'' : v} \; [Let_1]$$

$$\frac{\Delta, k : e_1 \Downarrow \Theta, k' : C\,\overline{a_i}^n@j \;\; j \leq k' \;\; fresh(p) \;\; \Theta \cup [p \mapsto (j, C\,\overline{a_i}^n)], k' : e[p/x_1] \Downarrow \Psi, k'' : v}{\Delta, k : \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e \Downarrow \Psi, k'' : v} \; [Let_2]$$

$$\frac{C = C_r \quad \Delta, k : e_r[\overline{a_j/x_{rj}}^{n_r}] \Downarrow \Theta, k' : v}{\Delta[p \mapsto (j, C\,\overline{a_i}^{n_r})], k : \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}^m \Downarrow \Theta, k' : v} \; [Case]$$

$$\frac{C = C_r \quad \Delta, k : e_r[\overline{a_j/x_{rj}}^{n_r}] \Downarrow \Theta, k' : v}{\Delta \cup [p \mapsto (j, C\,\overline{a_i}^{n_r})], k : \mathbf{case!}\ p\ \mathbf{of}\ \overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}^m \Downarrow \Theta, k' : v} \; [Case!]$$

Figura 2.2: Semántica operacional de *Core-Safe*

da puede ser leida, pero una vez que ha sido destruida o reutilizada no se puede acceder a la misma.

2.2.2. Semántica operacional de *Core-Safe*

En la Figura 2.2 se muestran las reglas que definen la semántica operacional para expresiones *Core-Safe*. Un juicio de la forma $\Delta, k : e \Downarrow \Theta, k' : v$ indica que puede reducirse la expresión e a la forma normal v bajo un montón Δ con $k+1$ regiones (desde 0 hasta k) y que además se produce un montón Θ con $k'+1$ regiones.

Un **montón** Δ es una función que asocia un puntero p con clausuras w de la forma $(j, C\,\overline{a_i}^n)$, indicando que la clausura se encuentra en la región j. Una **forma normal** es, o bien un valor básico c, o bien una construcción $C\,\overline{a_i}^n@j$ que se encuentra en la región j. Los parámetros a_i son valores básicos o punteros a otras clausuras. Los identificadores de región son en realidad números naturales.

Mediante la notación $\Delta\,[p \to w]$ se indica el montón Δ en el que puede encontrarse el vínculo $[p \to w]$. Por el contrario, la notación $\Delta \cup [p \to w]$ denota la unión disjunta del montón Δ con el enlace $[p \to w]$.

La semántica de un programa *Core-Safe* $d_1; \ldots; d_n; e$ (no mostrada en las reglas) es la semántica de la expresión principal e bajo un entorno Σ que contiene todas las definiciones de funciones $d_1; \ldots; d_n$.

Las reglas $[Lit]$ y $[Cons]$ tan sólo especifican que los valores básicos y construcciones son formas normales. La regla $[Cons]$ no crea ninguna clausura. Realmente las clausuras sólo se crean en la regla $[Let_2]$, que es la única que reserva memoria.

La regla $[Var_1]$ accede a la clausura correspondiente. La regla $[Var_2]$ realiza una copia completa de la ED apuntada por la variable p en una nueva región j. La función *copy* recorre los punteros de las posiciones recursivas de la estructura original, que se encuentra en la región l y crea una región j una copia de todas las clausuras recursivas excepto la clausura raíz $C\ \overline{a_i}^n$.

Si la función *copy* encuentra un puntero descolgado durante el recorrido, la regla entera fallaría y la derivación quedaría detenida en este punto. Si la copia tiene éxito, la expresión principal se convierte en la copia $C\ \overline{a'_i}^n$ de la clausura raíz, donde los punteros a_i de las posiciones recursivas han sido reemplazados por las correspondientes copias a'_i (que se encuentran en la región destino j). Los punteros que se encuentran en posiciones no recursivas son compartidos entre la estructura original y la copia. Por ejemplo, si la ED es una lista que contiene listas, la estructura creada por *copy* es una copia de la lista externa, mientras que las listas internas son compartidas por la lista original y su copia.

La regla $[Var_3]$ es similar a $[Var_1]$, con la diferencia de que el enlace $[p \rightarrow w]$ se elimina del montón Δ y por tanto, p no pertenece al dominio del montón resultante. Esto puede crear punteros descolgados en este último montón, ya que algunas clausuras pueden contener apariciones libres de p.

Las reglas $[App_1]$ y $[App_2]$ muestran cómo se crea una nueva región. El cuerpo de la función se ejecuta en un montón con $k + 2$ regiones. Esto es, el identificador de región *self* se asocia con la nueva región $k + 1$, de podrán crearse EDs en la misma o podrá ser pasada como parámetro en una llamada a función. Mediante la notación $\Theta|'_k$ se indica la restricción del montón Θ a las clausuras contenidas en las regiones desde 0 hasta k'. Antes de finalizar la ejecución de la función todas las clausuras creadas en la región $k' + 1$ son eliminadas. Esta acción puede resultar ser fuente de punteros potencialmente descolgados.

Las reglas $[Let_1]$ y $[Let_2]$ expresan el carácter impaciente del lenguaje: en primer lugar se reduce la expresión e_1 a forma normal y posteriormente se reduce la expresión principal e. Las apariciones de x_1 en esta última se reemplazan por la forma normal si es un valor básico, o por un puntero si es una construcción.

Por último, las reglas $[Case]$ y $[Case!]$ se diferencian en el hecho de que en esta última el enlace correspondiente al discriminante p es eliminado del montón. Esta acción puede crear también punteros descolgados.

2.2.3. Sintaxis *Full-Safe*

El lenguaje *Core-Safe* sólo contiene las construcciones indispensables para poder programar con el lenguaje SAFE. El hecho de ser un lenguaje tan sencillo permite sim-

prog	→	$def_1; \ldots; def_n; exp$	
def	→	*defFun*	{ecuación de definición de función, todas juntas}
		| **data constr** $\{\mathbf{id}\}^*$ @ $\{\mathbf{id}\}^+ = altDato_1; \ldots; altDato_n$	{definición de tipo de datos}
		| **id** :: *expTipo*	{tipo de una función, debe ir antes de las definiciones}
altDato	→	**constr @ id**	{constructor de alternativa sin parámetros}
		| (**constr** $\{expTipo\}^+$) **@ id**	{constructor de alternativa con parámetros}
		| (*expTipo* **cinfija** *expTipo*) **@ id**	{constructor de alternativa infijo}
expTipo	→	**id**	
		| **constr**	{constructor de tipo sin parámetros}
		| **constr** ($\{expTipo\}^+$) [!] @ $\{\mathbf{id}\}^+$	{constructor de tipo con parámetros}
		| (*expTipo* $\{, expTipo\}^+$) [!] **@ id**	{constructor de tipo para tuplas}
		| [*expTipo*] [!] **@ id**	{constructor de tipo para listas}
		| *expTipo* → *expTipo*	{separador de parámetros}
		| (*expTipo*)	
defFun	→	(*pat* | **id** $\{pat[!]\}^*$ | *pat*[!] **op** *pat*[!]) *der*	
der	→	= *exp* [**where** $\{def\{;def\}^*\}$]	
		| $\{\mid exp = exp\}^+$ [**where** $\{def\{;def\}^*\}$]	
pat	→	*lit*	
		| **id**	
		| [ ]	
		| (*pat* $\{, pat\}^+$)	
		| *pat* **cinfija** *pat*	
		| **constr** $\{pat\}^*$	
		| (*pat*)	
exp	→	*a*	{átomo: literal c o variable x}
		| x@r	{copia}
		| x!	{reutilización}
		| $(f\ \overline{exp_i}^n)$@r	{aplicación de función, que construye algo}
		| $(f\ \overline{exp_i}^n)$	{aplicación de función}
		| $(C\ \overline{exp_i}^n)$@r	{aplicación de constructor}
		| *exp* **cinfija** *exp*	{aplicación de constructor (infijo)}
		| *exp* **op** *exp*	{aplicación de operador infijo}
		| **let** $\overline{def_i}^n$ **in** *exp*	{recursivo, monomórfico}
		| **case** *exp* **of** $\overline{alt_i}^n$	{case de sólo lectura}
		| **case**! *exp* **of** $\overline{alt_i}^n$	{case destructivo}
		| (*exp*)	{expresión parentizada}
alt	→	*pat* → *expr*	

Figura 2.3: Sintaxis *Full-Safe*

plificar en gran medida el análisis de programas escritos en *Core-Safe*. Sin embargo, desde el punto de vista del programador resulta demasiado tedioso utilizar *Core-Safe* como lenguaje de implementación. Es necesaria una versión *edulcorada* del lenguaje que ofrezca más facilidades al programador. Esta versión del lenguaje SAFE es conocida como *Full-Safe*. Existe un proceso de transformación (ver Sección 2.4) durante la compilación, donde a partir de un programa *Full-Safe* se obtiene un programa *Core-Safe* equivalente.

La sintaxis *Full-Safe* está definida por la gramática de la Figura 2.3. Un programa sigue estando compuesto por una lista de definiciones y una expresión principal. No obstante, a diferencia de *Core-Safe*, además de funciones pueden definirse tipos de datos —mediante la construcción **data**— y signaturas de funciones.

La definición de tipos de dato es similar a su equivalente en Haskell. La única diferencia es la inclusión de *variables de región polimórficas* que en posteriores concreciones serán sustituídas por las regiones donde se alojen las construcciones del tipo que se está definiendo. También se permite el uso de constructoras infijas.

Con respecto a las expresiones de tipo, además de variables y tipos algebraicos se dispone de una sintaxis especial para tuplas $(t_1, \ldots, t_n)$ y para listas $[t]$. El símbolo (!) sólo tiene sentido cuando el tipo al que acompaña especifica el tipo de un parámetro de una función e indica que la estructura correspondiente a dicho parámetro será destruida durante la ejecución de la misma. Por otro lado, aunque se trata de un lenguaje de primer orden, la sintaxis no prohíbe expresar funciones dentro de tuplas o como parámetros de otras funciones. Con ello se permite una posible ampliación del lenguaje para incluir orden superior.

Dentro de una definición de función se permite la inclusión de *patrones* como parámetros de la misma y no sólo variables, como ocurría en *Core-Safe*. Por otro lado, pueden incorporarse *guardas* y cláusulas **where** en el lado derecho de una definición, al igual que en Haskell.

La sintaxis de las expresiones resulta similar a la de su análoga en *Core-Safe*. Como diferencias cabe señalar la inclusión de aplicación de constructores y funciones infijos y la posibilidad de introducir expresiones arbitrarias como parámetros de una constructora o función y como discriminante en un **case**(!).

2.3. Ejemplos

En esta sección se mostrarán algunos ejemplos de programas escritos en *Full-Safe* que utilizan las características de manejo de memoria: regiones, copia y destrucción explícita. En algunos casos también se incluirá el programa *Core-Safe* equivalente.

EJEMPLO 2 (Concatenación de listas)**.** *La siguiente función Full-Safe concatena las dos listas pasadas como parámetro.*

$$
\begin{array}{l}
concat :: [a]@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_2 \rightarrow [a]@\rho_2 \\
concat\ [\,]\ ys\ @r = ys \\
concat\ (x : xs)\ ys\ @r = (x : concat\ xs\ ys\ @r)@r
\end{array}
$$

Los dos primeros parámetros son listas que se alojan en las regiones de tipo ρ_1 y ρ_2, respectivamente. Esto implica que las dos listas de entrada podrían estar situadas en regiones distintas. El tercer parámetro indica la región donde se situará la lista resultado (ρ_2). Con ello se obliga a que el resultado resida en la misma región que la segunda lista pasada como prámetro. Esto viene impuesto por el hecho de que en la primera ecuación que define la función (caso en que el primer parámetro es $[\,]$) se devuelve directamente el segundo parámetro como resultado.

En la lista de parámetros de cada ecuación se incluye una variable r tras el símbolo @. Esta variable representa la región de destino y se propaga en las llamadas recursivas a la función concat. Además se utiliza para indicar la región donde se situará la nueva clausura creada que será devuelta como resultado.

La Figura 2.4 muestra un ejemplo de ejecución de concat para las listas $[1,2]$ y $[3,4]$. La primera de ellas reside en la región r_1 y la segunda en r_2. En esta figura puede verse la evolución de la región r_2, que pasa a incluir las nuevas celdas creadas. La región r_1 no se modifica.

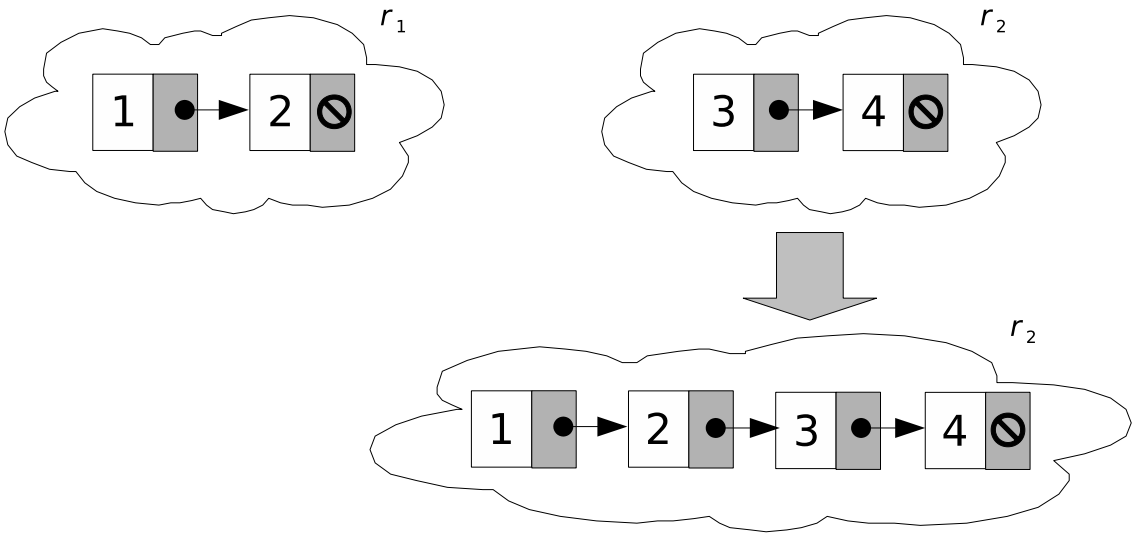

Figura 2.4: Concatenación de dos listas

El programa Core-Safe equivalente se muestra a continuación:

$$
\begin{array}{l}
concat\ zs\ ys\ @r = \textbf{case}\ zs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \rightarrow ys \\
\qquad\qquad\qquad\qquad (x : xs) \rightarrow \textbf{let}\ x_1 = concat\ xs\ ys\ @r\ \textbf{in}\ (x : x_1)@r
\end{array}
$$

Como puede observarse, las distintas ecuaciones han sido reemplazadas por una en la que se realiza una distinción de casos explícita mediante un **case**. *Por otro lado, la llamada recursiva que se encontraba dentro de una construcción ha sido sustituida por una variable x_1 y se ha incorporado el* **let** *correspondiente.*

Esta definición impone la restricción de que la lista resultante ha de residir en la misma región que la lista pasada como segundo parámetro, ya que en el caso base se reutiliza esta última. Si se quiere que la lista resultado se aloje en una región distinta es necesario utilizar la operación de copia, tal como se muestra en la siguiente definición de $concat'$:

$$
\begin{array}{l}
concat' :: \rho_2 \neq \rho_3\,.\,[a]@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_3 \rightarrow [a]@\rho_3 \\
concat'\ [\,]\ ys\ @r = ys@r \\
concat'\ (x : xs)\ ys\ @r = (x : concat'\ xs\ ys\ @r)@r
\end{array}
$$

En el caso base ya no se devuelve la estructura apuntada por ys, sino una copia de la misma que se alojará en la región de destino r. La ED original y su copia han de residir en regiones distintas, por lo que se incluye la restricción $\rho_2 \neq \rho_3$ en el tipo correspondiente.

EJEMPLO 3 (Concatenación de listas destructiva)**.** *El ejemplo anterior conservaba las listas pasadas como parámetro. La siguiente versión de la concatenación, concatD, libera la memoria ocupada por la lista pasada como primer parámetro:*

$$
\begin{array}{l}
concatD :: [a]!@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2 \\
concatD\ [\,]!\ ys\ @r = ys \\
concatD\ (x : xs)!\ ys\ @r = (x : concatD\ xs\ ys\ @r)@r
\end{array}
$$

La única diferencia con respecto a la definición original consiste en la inclusión del signo ! tras el primer parámetro. Esto provocará que en el programa Core-Safe equivalente se utilice la versión destructiva de **case***:*

$$
\begin{array}{l}
concatD\ zs\ ys\ @r = \textbf{case}!\ zs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \to ys \\
\qquad\qquad\qquad\qquad (x : xs) \to \textbf{let}\ x_1 = concatD\ xs\ ys\ @r\ \textbf{in}\ (x : x_1)@r
\end{array}
$$

En efecto, la primera celda cabeza-resto de la lista quedará liberada tras realizar la distinción de casos. El resto de la estructura cons-nil de la misma se liberará en las siguientes llamadas recursivas a concatD.

Cabe señalar que el hecho de destruir la primera lista queda reflejado en la signatura de la función concatD: el primer parámetro tiene tipo $[a]!@\rho_1$, donde el signo ! indica que se trata de un tipo condenado.

Puede utilizarse la función concatD para implementar otra versión de la concatenación no destructiva de listas:

$$
\begin{array}{l}
concat''\ xs\ ys\ @r = concatD\ zs\ ys \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ zs = xs@self
\end{array}
$$

En este caso se realiza una copia temporal en la región de trabajo self que será posteriormente destruida en la llamada a concatD.

EJEMPLO 4 (Inversión de una lista)**.** *Puede utilizarse la definición concat para implementar la función que invierte los elementos de una lista de partida, liberando esta última.*

$$
\begin{array}{l}
reverseD :: [a]!@\rho_1 \to \rho_2 \to [a]@\rho_2 \\
reverseD\ [\,]!\ @r = [\,] \\
reverseD\ (x : xs)!\ @r = concat\ (reverseD\ xs\ @r)\ ((x : [\,]@r)@r)@r
\end{array}
$$

Esta función tiene coste en tiempo de $\mathcal{O}(n^2)$, donde n es la longitud de la lista de partida. La siguiente implementación de reverse′ con parámetro acumulador mejora este coste:

$$
\begin{array}{l}
reverse' :: [a]!@\rho_1 \to \rho_2 \to [a]@\rho_2 \\
reverse'\ xs\ @r = revauxD\ xs\ ([\,]@r)\ @r \\
\\
revauxD :: [a]!@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2 \\
revauxD\ [\,]!\ ys\ @r = ys \\
revauxD\ (x : xs)!\ ys\ @r = revauxD\ xs\ ((x : ys)@r)\ @r
\end{array}
$$

Con esta función se obtiene un coste en tiempo lineal con respecto a la longitud de la lista de partida. El coste en memoria es constante, ya que a medida que se libera una celda de entrada se construye una celda del resultado. A continuación se muestra la versión Core-Safe de la función revauxD:

$$
\begin{array}{l}
\mathit{revauxD}\ \mathit{zs}\ \mathit{ys}\ @r = \mathbf{case}!\ \mathit{zs}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad [\,] \rightarrow \mathit{ys} \\
\qquad\qquad\qquad\qquad (x : \mathit{xs}) \rightarrow \mathbf{let}\ x_1 = (x : \mathit{ys})@r\ \mathbf{in}\ \mathit{revauxD}\ \mathit{xs}\ x_1\ @r
\end{array}
$$

El uso de destrucción explícita en el lenguaje no sólo permite prescindir de un mecanismo de recolección de basura, sino que también facilita el razonamiento sobre el espacio en memoria adicional que se necesita para ejecutar una función. La siguiente implementación del algoritmo *Mergesort* es un ejemplo de ello:

EJEMPLO 5 (Mergesort)**.** *En primer lugar se definirán las habituales funciones auxiliares para dividir una lista de entrada y mezclar dos listas ordenadas en una sóla lista ordenada:*

$$
\begin{array}{l}
\mathit{splitD} :: \mathit{Int} \rightarrow [a]!@\rho \rightarrow \rho \rightarrow ([a]@\rho, [a]@\rho)@\rho \\
\mathit{splitD}\ 0\ \mathit{xs}!\ @r = ([\,]@r, \mathit{xs}!)@r \\
\mathit{splitD}\ n\ [\,]!\ @r = ([\,]@r, [\,]@r)@r \\
\mathit{splitD}\ n\ (x : \mathit{xs})!\ @r = ((x : \mathit{xs}_1)@r, \mathit{xs}_2)@r \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{where}}\ (\mathit{xs}_1, \mathit{xs}_2)! = \mathit{splitD}\ (n-1)\ \mathit{xs}\ @r
\end{array}
$$

Al igual que en ejemplos anteriores, no hay muchas diferencias con respecto a una versión implementada en un lenguaje funcional puro. En el caso base ($n = 0$) se reutiliza la lista xs como parte del resultado. En el caso recursivo se utiliza ajuste de patrones destructivo sobre el segundo argumento. También se añade @r donde resulte necesario.

$$
\begin{array}{l}
\mathit{mergeD} :: [a]!@\rho \rightarrow [a]!@\rho \rightarrow \rho \rightarrow [a]@\rho \\
\mathit{mergeD}\ [\,]!\ \mathit{ys}!\ @r = \mathit{ys}! \\
\mathit{mergeD}\ (x : \mathit{xs})!\ \ [\,]!\ @r = (x : \mathit{xs}!)@r \\
\mathit{mergeD}\ (x : \mathit{xs})!\ \ (y : \mathit{ys})!\ @r \\
\qquad\qquad\qquad\qquad\quad |\ \ x \leq y\ \ =\ \ (x : \mathit{mergeD}\ \mathit{xs}\ ((y : \mathit{ys}!)@r)\ @r)@r \\
\qquad\qquad\qquad\qquad\quad |\ \ x > y\ \ =\ \ (y : \mathit{mergeD}\ ((x : \mathit{xs}!)@r)\ \mathit{ys}\ @r)@r
\end{array}
$$

En las llamadas recursivas a mergeD los parámetros son las listas originales. Sin embargo no puede hacerse referencia a dichas listas, ya que han sido condenadas mediante un ajuste de patrones destructivo. Por tanto, es necesario reconstruir la lista mediante la reutilización de sus componentes.

Con ayuda de estas funciones auxiliares puede definirse la función msortD del siguiente modo:

```
msortD :: [a]!@ρ → ρ → [a]@ρ
msortD xs @r
          |  n ≤ 1   =  xs!
          |  n > 1   =  mergeD (msortD xs1 @r) (msortD xs2 @r)@r
             where  (xs1, xs2)!  =  splitD (n div 2) xs @r
                    n            =  length xs
```

Puede demostrarse que esta versión consume un espacio de memoria adicional constante. A continuación se muestra de un modo informal el razonamiento por inducción sobre la lista de entrada:

- *En el caso base se reutiliza la lista de entrada. Por tanto, no se necesita espacio de memoria adicional.*
- *En el caso recursivo, puede demostrarse fácilmente que las funciones splitD y mergeD consumen espacio de memoria constante.*

  En primer lugar se realiza una llamada a splitD con xs. Esta función destruye xs y construye un par de listas xs_1 y xs_2 tales que $length(xs) = length(xs_1) + length(xs_2)$ (esto puede demostrarse por inducción). Por tanto, el único espacio adicional necesario es el ocupado por el constructor de pares, que es posteriormente liberado para acceder a las componentes.

  A continuación se realizan dos llamadas recursivas a mergeD que, por hipótesis de inducción, tienen un coste en espacio adicional de memoria constante. Las listas resultado tienen la misma longitud que las originales, lo cual puede demostrarse por inducción.

  Por último se aplica la función mergeD que destruye las listas pasadas como argumento y produce una única lista cuya longitud es $length(xs'_1) + length(xs'_2)$. Por tanto, no se requiere espacio de memoria adicional.

En el Capítulo 5 se mostrará el comportamiento del algoritmo de inferencia presentado en este trabajo con esta implementación de *Mergesort*.

Todos los ejemplos expuestos hasta el momento trabajaban sobre el tipo de datos lista. A continuación se muestra un ejemplo en el que se declaran los árboles de búsqueda binarios junto con su correspondiente función de inserción.

EJEMPLO 6 (Árboles de búsqueda binarios)**.** *Se define el TAD árbol de búsqueda binario del siguiente modo:*

```
data BSTree a @r = Empty @r | Node (BSTree a @r) a (BSTree a @r) @r
```

El tipo de datos es polimórfico no sólo en el tipo de elementos contenidos en el árbol, sino en la región donde se encuentra alojado. Los hijos recursivos (primer y tercer argumento del constructor Node) deberán pertenecer a la misma región.

La función insertD inserta un elemento en un árbol de búsqueda binario de enteros:

$$
\begin{array}{l}
insertD :: Int \rightarrow BSTree\ Int\ @\rho \rightarrow \rho \rightarrow BSTree\ Int\ @\rho \\
insertD\ x\ Empty!\ @r = (Node\ Empty@r\ x\ Empty@r)@r \\
insertD\ x\ (Node\ i\ y\ d)! \\
\qquad\qquad \mid\ x < y \ = \ (Node\ (insertD\ x\ i)@r\ y\ d!)@r \\
\qquad\qquad \mid\ x = y \ = \ (Node\ i!\ y\ d!)@r \\
\qquad\qquad \mid\ x > y \ = \ (Node\ i!\ y\ (insertD\ x\ d)@r)@r
\end{array}
$$

Mediante un ajuste de patrones destructivo se libera la raíz del árbol. No obstante, siempre se reutilizará al menos uno de sus hijos recursivos. Durante una inserción, todo el árbol se reutiliza excepto el camino desde la raíz del mismo hasta el elemento insertado. Si el elemento a insertar ya se encontraba en el árbol (caso $x = y$) es necesario volver a reconstruir el nodo que fue destruido.

2.4. Visión general del compilador SAFE

El compilador para el lenguaje SAFE está aún en desarrollo. Actualmente se encuentra implementada en Haskell la parte frontal del mismo, que realiza los análisis estáticos necesarios para garantizar que un programa *Full-Safe* no realizará en ejecución accesos a estructuras de datos ya liberadas.

En la Figura 2.5 se muestran las fases de compilación actualmente implementadas. La inferencia de tipos Hindley-Milner, el análisis de compartición y la inferencia de tipos SAFE fueron implementados por el autor. Este trabajo contendrá detalles de implementación relativos a estos análisis. El resto de las fases fueron realizadas por alumnos de Ingeniería Informática en el marco de un proyecto de la asignatura Sistemas Informáticos. En [CLL06] puede encontrarse más información relacionada con las mismas.

A continuación se ofrece una breve descripción de las distintas fases actualmente implementadas:

- **Analizador léxico/sintáctico:** Recibe como entrada un programa escrito en *Full-Safe* (el cual viene dado en forma de cadena de caracteres) y obtiene una representación de su sintaxis en forma de árbol abstracto. Para la implementación de esta fase se han utilizando herramientas estándar, como *Alex* y *Happy* [DJM05, MG01].

- **Restricciones contextuales:** Se encarga de realizar determinadas comprobaciones en relación con el ámbito de los identificadores. De este modo se garantiza que los identificadores están bien definidos. Por otra parte se renombran todas las variables para que no existan dos variables ligadas distintas con el mismo nombre, lo cual podría presentar problemas en análisis posteriores. De este modo cada variable del programa recibe un nombre único.

- **Inferencia de tipos Hindley-Milner:** Realiza un recorrido del árbol abstracto, decorando cada elemento con su tipo correspondiente, generando ecuaciones de

unificación entre tipos y resolviéndolas a continuación. En el Capítulo 3 se detalla el proceso completo.

- **Transformación a *Core-Safe*:** A partir del programa *Full-Safe* se obtiene un programa *Core-Safe* equivalente, preservando además la información de tipos obtenida en la fase anterior. Las transformaciones realizadas están basadas en [Jon87].
- **Análisis de compartición:** Permite aproximar las posibles relaciones de compartición entre las variables de una definición. Este análisis se detalla en el Capítulo 4 y sus resultados serán necesarios en la siguiente fase de compilación.
- **Inferencia de tipos SAFE:** Se encarga de comprobar si el programa admite un tipo válido SAFE y, en su caso, de inferirlo. Dicho tipo garantiza que en tiempo de ejecución no se realizarán accesos a punteros descolgados en la memoria. Este es, por ahora, el análisis más complejo implementado en el compilador. Se encuentra descrito en el Capítulo 5.

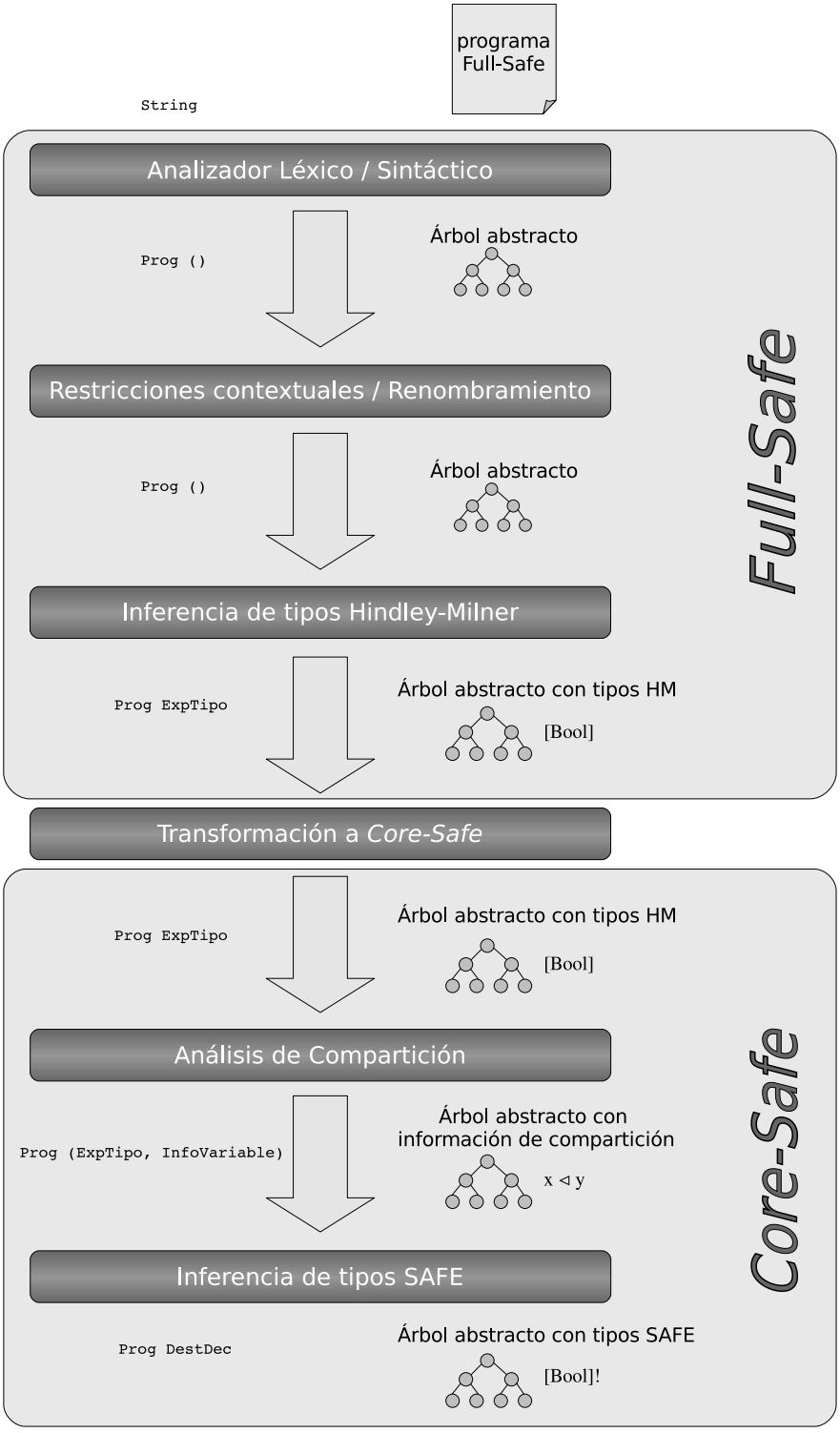

Figura 2.5: Fases de compilación

Capítulo 3

Inferencia de tipos Hindley-Milner

En este capítulo se abordará el problema de la reconstrucción de tipos (en tiempo de compilación) de las definiciones contenidas en un programa escrito en *Full-Safe*. Suponemos que el programa ya ha sido analizado sintácticamente y se ha obtenido su árbol abstracto correspondiente. Del mismo modo, se suponen realizadas todas las comprobaciones semánticas que forman parte de la definición del lenguaje, como por ejemplo, que toda aparición de uso de una variable se encuentra en ámbito o que todas las ecuaciones que definen una función aparecen consecutivas. Cada elemento del árbol abstracto quedará decorado al final del análisis con su correspondiente tipo.

Uno de los algoritmos utilizados para realizar una inferencia de tipos recibe el nombre de **inferencia de tipos Hindley-Milner**. Este algoritmo puede expresarse en dos fases: generación de ecuaciones entre tipos y resolución de las mismas. Puede encontrase una explicación detallada del mismo en [Pie02].

En las siguientes secciones se describirá tanto el algoritmo de inferencia general aplicado al lenguaje *Full-Safe*, así como determinadas fases específicas de este lenguaje para el tratamiento de las regiones. No obstante, no se realizará en esta fase de compilación ninguna comprobación relativa a la destrucción segura de estructuras de datos. Los tipos resultantes de dicha comprobación (conocidos como *tipos SAFE*) y el algoritmo correspondiente serán tratados en el Capítulo 5.

3.1. Expresiones de tipo

En la Figura 3.1 podemos encontrar la sintaxis de los tipos Hindley-Milner que serán inferidos por el algoritmo expuesto en este capítulo y establecidos como decoración de los distintos elementos del árbol abstracto.

Al tratarse de un lenguaje de primer orden, en la sintaxis de tipos de SAFE distinguimos entre tipos funcionales *tf* y tipos no funcionales *t*. Los tipos funcionales tienen la forma $t_1 \to \cdots \to t_n \to t$, o bien, $t_1 \to \cdots \to t_n \to \rho \to t$, donde $t_1, \ldots, t_n, t$ son tipos no funcionales y ρ es una variable de tipo de región. En el primer caso no se construye ninguna estructura de datos y en el segundo caso se construye una estructura de datos que será guardada en una región de memoria de tipo ρ.

$$
\begin{array}{rcll}
t & \rightarrow & b & \{\text{tipo básico}\} \\
 & & \mid a & \{\text{variable de tipo}\} \\
 & & \mid T\ \overline{t_i}^n\ @\overline{\rho_i}^m & \{\text{tipo algebraico}\} \\
 & & & \\
b & \rightarrow & Int & \{\text{tipo básico entero}\} \\
 & & \mid Bool & \{\text{tipo básico booleano}\} \\
 & & & \\
tf & \rightarrow & t & \{\text{tipo resultado}\} \\
 & & \mid \rho \rightarrow t & \{\text{región destino y tipo resultado}\} \\
 & & \mid t \rightarrow tf & \{\text{parámetro de entrada}\} \\
 & & & \\
tfr & \rightarrow & \overline{\rho_i \neq \rho_i'}^n\ .\ tf & \{\text{tipo restringido}\}
\end{array}
$$

Figura 3.1: Expresiones de tipo

Por su parte, los tipos no funcionales pueden ser tipos *básicos* o *algebraicos*. Los valores de tipos básicos nunca se almacenarán en el montón (*heap*) y, por tanto, no pueden realizarse operaciones de construcción, copia o liberación de memoria sobre variables de estos tipos. Dentro de este grupo se incluyen los valores enteros y booleanos (de tipo *Int* y *Bool*, respectivamente). Por otro lado, los valores de tipos algebraicos se almacenan en regiones del montón que pueden ser liberadas. Las *variables de tipo* pueden ser utilizadas en tipos polimórficos y se refieren tanto a tipos básicos como algebraicos.

Asociadas a las funciones pueden aparecer tipos *restringidos*, esto es, tipos funcionales con restricciones entre variables de región adjuntas. Esta clase de tipos surge a partir de la necesidad de imponer regiones distintas entre origen y destino cuando se copia una estructura de datos.

En las secciones siguientes se utilizarán las letras $s, t, \ldots$ para designar los distintos tipos, sin hacer distinción entre funcionales y no funcionales. El contexto en el que aparezcan determinará si pertenecen a una u otra clase. Por ejemplo, en expresiones de la forma $t_1 \rightarrow t_2$ se considera que t_1 es un tipo no funcional y t_2 es un tipo funcional.

3.2. Generación de ecuaciones

La inferencia de tipos se realiza de forma independiente para cada conjunto de ecuaciones que conforman la definición de una función. La primera fase de esta inferencia es el recorrido del árbol abstracto correspondiente a dichas ecuaciones, la introducción de ecuaciones de unificación entre tipos necesarias y la decoración del mismo.

- Si puede deducirse el tipo de un elemento del árbol abstracto sin necesidad de recurrir al proceso de resolución de ecuaciones (siguiente fase del algoritmo), entonces dicho elemento queda decorado con el tipo definitivo. Éste es el caso de

los literales enteros y booleanos, que siempre tendrán tipo *Int* y *Bool*, respectivamente.

- Si es necesario el procedimiento posterior de resolución de ecuaciones para obtener el tipo de un elemento del árbol abstracto (como por ejemplo en una llamada a una función, cuyo resultado puede tener un tipo que en este momento de la inferencia es desconocido), se decorará dicho elemento con una variable de tipo que será sustituida por el tipo definitivo en una fase posterior.

Para generar las ecuaciones de tipos es necesario mantener un conjunto A de suposiciones de tipo. Este conjunto consta de vínculos $[x : t]$, donde x es el nombre de una variable de programa o de una función y t es un tipo, y mantiene los nombres disponibles en ámbito a medida que se recorre cada definición. El recorrido de una expresión obtiene como resultado:

- Un conjunto de ecuaciones E de la forma $t_1 = t_2$, que indica que los tipos t_1 y t_2 deben unificar.
- Un conjunto de restricciones C de la forma $t = \mathit{isData}(t', \rho)$, cuyo significado será descrito en la Sección 3.4.
- Un conjunto de desigualdades I de la forma $\rho_1 \neq \rho_2$, indicando que las regiones representadas por ρ_1 y ρ_2 deben ser diferentes.

Adicionalmente, durante el recorrido de patrones y definiciones, puede generarse un conjunto A' de suposiciones nuevas que incorpora las variables recién introducidas en el ámbito actual.

En las reglas de tipado para expresiones encontraremos juicios de la forma:

$$A \vdash e : a \mid E, C, I \qquad \text{o bien} \qquad A \vdash e : t \mid E, C, I$$

que indican que a partir de un conjunto de suposiciones A, el análisis de la expresión e y todas sus subexpresiones da lugar al conjunto de ecuaciones E, restricciones C y desigualdades I. Además, la expresión e quedará decorada, en el primer caso, por una variable de tipo *fresca* y en el segundo caso por el tipo t indicado.

Por otro lado, en las reglas para patrones los juicios tienen la siguiente forma:

$$A \vdash_p p : a \mid A', E \qquad \text{o bien} \qquad A \vdash_p p : t \mid A', E$$

donde se indica que se ha generado un conjunto A' de suposiciones nuevas y un conjunto E de ecuaciones al analizar el patrón p. Por último, las declaraciones de la forma $p = e$ incluidas dentro de un **let** o **where** no se decoran y, por tanto, los juicios resultantes son de la forma:

$$A \vdash_p p = e \mid A', E$$

Asímismo, utilizaremos la notación $A \vdash f \triangleleft t$ para indicar que t es una concreción fresca generada a partir del esquema de la función f contenido en A. Para constructoras de datos C, tendremos $A \vdash C \triangleleft t$, con el mismo significado.

$$\frac{}{A \vdash i : Int \mid \emptyset, \emptyset, \emptyset} \qquad \frac{}{A \vdash b : Bool \mid \emptyset, \emptyset, \emptyset} \qquad \frac{A(x) = t}{A \vdash x(!) : t \mid \emptyset, \emptyset, \emptyset}$$

$$\frac{A(x) = t \quad A(r) = \rho}{A \vdash x@r : a \mid \emptyset, \{a = isData(t, \rho)\}, \emptyset}$$

$$\frac{A \vdash f \triangleleft I.t_0 \quad \forall i \,.\, A \vdash e_i : t_i \mid E_i, C_i, I_i \quad A(r) = \rho}{A \vdash (f\ \overline{e_i}^n)@r : a \mid (\bigcup_i E_i) \cup \{t_0 = t_1 \to \cdots \to t_n \to \rho \to a\}, \bigcup_i C_i, (\bigcup_i I_i) \cup I}$$

$$\frac{A \vdash C \triangleleft t_0 \quad \forall i \,.\, A \vdash e_i : t_i \mid E_i, C_i, I_i \quad A(r) = \rho}{A \vdash (C\ \overline{e_i}^n)@r : a \mid (\bigcup_i E_i) \cup \{t_0 = t_1 \to \cdots \to t_n \to \rho \to a\}, \bigcup_i C_i, \bigcup_i I_i}$$

$$\frac{A \vdash_p p_1 = e_1 \mid A_1, E_1 \quad A \cup A_1 \vdash e : t \mid E, C, I}{A \vdash \textbf{let}\ p_1 = e_1\ \textbf{in}\ e : t \mid E_1 \cup E, C_1 \cup C, I_1 \cup I}$$

$$\frac{A \vdash e : t \mid E, C, I \quad \forall i \left(A \vdash_p p_i : t_i \mid A_i, E_i \ \wedge\ A \cup A_i \vdash e_i : t_i' \mid E_i', C_i', I_i'\right)}{\begin{array}{r} A \vdash \textbf{case}(!)\ e\ \textbf{of}\ \overline{p_i \to e_i}^n : t_1' \mid \quad (\bigcup_i E_i) \cup (\bigcup_i E_i') \cup E \cup E_x, \\ (\bigcup_i C_i) \cup (\bigcup_i C_i') \cup C, \\ (\bigcup_i I_i) \cup (\bigcup_i I_i') \cup I \end{array}}$$

$$\textbf{donde}\ E_x = \{\ \ t = t_1,\ \ t_1 = t_2,\ \ \ldots,\ \ t_{n-1} = t_n,\ \ t_1' = t_2',\ \ \ldots\ \ t_{n-1}' = t_n'\ \ \}$$

Figura 3.2: Reglas de generación de ecuaciones para expresiones

En la Figura 3.2 podemos encontrar las reglas de generación de ecuaciones para expresiones.

Los literales (enteros y booleanos) pueden decorarse directamente con el tipo correspondiente sin generar ninguna ecuación o restricción.

Para decorar una variable basta con acceder a su tipo contenido en el conjunto de suposiciones. El caso en que se hace copia de la estructura a la que apunta una variable es similar al anterior, pero esta vez la decoración es una variable fresca y se genera una restricción *isData* (ver Sección 3.4 para más detalles).

La generación de ecuaciones para una llamada a una función f requiere, en primer lugar, obtener una instancia fresca del tipo de f y, por otro lado, los tipos de cada uno de los parámetros y de la región de salida. El tipo de cada parámetro de t_0 obtenido a partir de las suposiciones debe unificar con el tipo de cada parámetro pasado a la llamada, y el resultado de la llamada (resultado del tipo t_0) debe unificar con la variable fresca a con la que se decora la llamada. Además, si el tipo de f obtenido a partir de las suposiciones lleva adjunto un conjunto I de desigualdades entre variables de región, debemos añadir dicho conjunto a las desigualdades del resultado. El caso de

$$\frac{}{A \vdash_p i : Int \mid \emptyset, \emptyset} \qquad \frac{}{A \vdash_p b : Bool \mid \emptyset, \emptyset} \qquad \frac{}{A \vdash_p x : a \mid [x : a], \emptyset}$$

$$\frac{A \vdash C \lhd t_0 \quad \forall i \,.\, A \vdash p_i : t_i \mid A_i, E_i \quad fresh(\rho)}{A \vdash_p C\ \overline{p_i}^n : a \mid (\uplus_i A_i), (\bigcup_i E_i) \cup \{t_0 = t_1 \to \cdots \to t_n \to \rho \to a\}}$$

Figura 3.3: Reglas de generación de ecuaciones para patrones

construcción de una estructura de datos es similar al de una llamada a una función, salvo que en este caso no tenemos desigualdades que añadir.

Para el análisis de expresiones **let** basta con analizar las expresiones auxiliar y principal por separado y reunir los resultados de cada una. Las suposiciones nuevas sobre p_1 generadas en la expresión auxiliar deben tenerse en cuenta en el momento de recorrer e, dado que x_1 pasa a estar en ámbito. Si existen varias definiciones auxiliares en la expresión **let**, se realizará la generación para cada definición por separado. Con el conjunto de suposiciones acumuladas se pasa a generar las ecuaciones para e. En este caso el esquema es similar al utilizado en cláusulas **where** (Figura 3.3).

En el caso de expresiones **case** y **case**! se examina cada alternativa por separado y se reunen las ecuaciones, restricciones y desigualdades obtenidas junto con los resultados de examinar la expresión discrimante e. Además se añade el conjunto de ecuaciones E_x que obliga a que el tipo del discriminante sea igual al tipo de cada uno de los patrones de las alternativas y que los tipos de las expresiones de cada alternativa sean iguales.

Las reglas de generación de ecuaciones relacionadas con patrones (Figura 3.3) tienen un aspecto parecido a las anteriores. La única diferencia a remarcar consiste en el hecho de que ahora la aparición de las variables es de definición, no de uso. Por tanto, en lugar de consultar el tipo de la variable en el conjunto de suposiciones, se le debe asignar un tipo aún desconocido (variable de tipo fresca a) y generar una suposición nueva.

En la sintaxis *Full-Safe* podemos tener ecuaciones guardadas por expresiones booleanas, cláusulas **where** y definiciones en las que la parte izquierda es un patrón. La Figura 3.4 define las ecuaciones que deben generarse para las definiciones.

La primera regla trata el caso de definiciones auxiliares introducidas mediante una cláusula **let** o **where**. Los tipos de ambos lados de la definición deben unificar. La segunda regla se encarga de las definiciones guardadas por expresiones booleanas: las guardas han de tener tipo booleano y todas las alternativas han de tener el mismo tipo. La tercera regla especifica el tratamiento de la parte derecha de una definición con cláusula **where**. En este caso se examina cada definición auxiliar por separado, ampliando progresivamente el conjunto de suposiciones por cada nueva definición; posteriormente se analiza la expresión principal. La última regla se encarga de las definiciones de funciones: en primer lugar se examinan los patrones y se generan las suposiciones correspondientes, introduciendo además el tipo de la variable de región asociada a *self*. A continuación se analiza la parte derecha de la definición.

$$\frac{A \vdash e : t \mid E, C, I \quad A \vdash_p p : t' \mid A', E'}{A \vdash_p p = e \mid A', E \cup E' \cup \{t = t'\}}$$

$$\frac{\forall i.\, \left(A \vdash e_i : t_i \mid E_i \quad A \vdash e'_i : t'_i \mid E'_i\right)}{A \vdash \overline{|e_i = e'_i}^n : t'_1 \mid (\bigcup_i E_i) \cup (\bigcup_i E'_i) \cup E}$$
$$\textbf{donde } E = \{t_1 = Bool, \ldots, t_n = Bool, t'_1 = t'_2, \ldots, t'_{n-1} = t'_n\}$$

$$\frac{\begin{array}{rcl} A_0 & \vdash_p & p_1 = e_1 : t_1 \mid A_1, E_1 \\ A_0 \uplus A_1 & \vdash_p & p_2 = e_2 : t_2 \mid A_2, E_2 \\ \ldots & & \\ \biguplus_{i=0}^{n-1} A_i & \vdash_p & p_n = e_n : t_n \mid A_n, E_n \\ \biguplus_{i=0}^{n} A_i & \vdash & e : t \mid E \end{array}}{A_0 \vdash_p e \textbf{ where } \overline{p_i = e_i}^n \mid \bigcup_i A_i, (\bigcup_i E_i) \cup E}$$

$$\frac{\begin{array}{c} A \vdash f : a_0 \mid \emptyset, \emptyset, \emptyset \quad \mathit{fresh}(\rho) \quad \mathit{fresh}(\rho') \\ \forall i\,.\,(A \vdash_p p_i : t_i \mid A_i, E_i) \\ A \cup (\bigcup_i A_i) \cup [r : \rho] \cup [\mathit{self} : \rho'] \cup [f : a_0] \vdash e : t \mid E, C, I \end{array}}{A \vdash_p f\ \overline{p_i}^n\ @r = der : t \mid [f : a_0], E \cup (\bigcup_i E_i) \cup E', C, I}$$
$$\textbf{donde } E' = \{a_0 = t_1 \to \cdots \to t_n \to \rho \to t\}$$

Figura 3.4: Reglas de generación de ecuaciones para definiciones

3.3. Resolución de ecuaciones

El siguiente paso en la inferencia de tipos Hindley-Milner es la resolución de las ecuaciones generadas en la fase anterior, obteniendo como solución una sustitución θ que asocia a cada variable de tipo un tipo arbitrario no restringido.

Para resolver las ecuaciones utilizaremos un algoritmo de unificación sintáctica entre tipos que, a partir de un conjunto de ecuaciones, construye incrementalmente el unificador más general. El algoritmo propuesto en esta sección está basado en [CB83], una mejora cuadrática del algoritmo exponencial original de Robinson [Rob65].

El método de resolución de ecuaciones consiste en una transformación progresiva del conjunto E de ecuaciones. Para ello definimos una relación ($\Longrightarrow$) sobre pares de la forma (E, θ), donde E es un conjunto de ecuaciones y θ es una sustitución. Definimos ($\Longrightarrow$) como la menor relación que cumple las reglas contempladas en la Figura 3.5.

El algoritmo de unificación comenzará con la sustitución vacía $\emptyset$ y el conjunto de ecuaciones a resolver. Ambos se transformarán en sucesivos pasos de resolución ($\Longrightarrow$) hasta obtener el conjunto vacío de ecuaciones. Es decir, si ($\Longrightarrow^*$) es el cierre reflexivo-

$$\boxed{E, \theta \Longrightarrow E', \theta'}$$

$$\begin{array}{rcll}
\{t = t\} \uplus E, \theta & \Longrightarrow & E, \theta & \\
\{a = t\} \uplus E, \theta & \Longrightarrow & E\,[t/a]\,, [a \mapsto t] \circ \theta & \text{si } a \notin FV(t) \\
\{t = a\} \uplus E, \theta & \Longrightarrow & E\,[t/a]\,, [a \mapsto t] \circ \theta & \text{si } a \notin FV(t) \\
\{s_1 \to s_2 = t_1 \to t_2\} \uplus E, \theta & \Longrightarrow & E \cup \{s_1 = t_1, s_2 = t_2\}, \theta & \\
\{T\ s_1 \ldots s_n @ \rho_1 \ldots \rho_m = T\ t_1 \ldots t_n @ \rho'_1 \ldots \rho'_m\} \uplus E, \theta & & & \\
\Longrightarrow E \cup \{s_1 = t_1, \ldots, s_n = t_n, \rho_1 = \rho'_1, \ldots, \rho_m = \rho'_m\}, \theta & & &
\end{array}$$

Figura 3.5: Reglas de unificación de tipos

transitivo de ($\Longrightarrow$), la solución (si existe) del sistema de ecuaciones E es la sustitución θ que cumple:

$$E, \emptyset \Longrightarrow^* \emptyset, \theta$$

En las reglas expresadas en la Figura 3.5 existe un no determinismo al elegir la ecuación a resolver. En la implementación se resuelve mediante el uso de una lista para almacenar las ecuaciones. Se resolverá la ecuación contenida en la cabeza de dicha lista. Si no es posible aplicar un paso de resolución a un conjunto de ecuaciones E, el proceso de resolución de ecuaciones devolverá fallo. Esto puede ocurrir en dos casos:

- No se cumple la condición $a \notin FV(t)$ en las ecuaciones de la forma $a = t$ y $t = a$.
- Se intenta unificar una ecuación de la forma

  $$T\ s_1 \ldots s_n @ \rho_1 \ldots \rho_m = T'\ t_1 \ldots t_{n'} @ \rho'_1 \ldots \rho'_{m'}$$

  donde $T = T'$.

3.4. Comprobación de restricciones *isData*

Todo lo explicado en las secciones anteriores se corresponde con el algoritmo de inferencia de tipos Hindley-Milner que no difiere en gran medida del algoritmo utilizado para inferir los tipos de un lenguaje funcional como Haskell. No obstante, como se ha visto anteriormente, el sistema de tipos permite añadir restricciones de la forma $\rho_1 \neq \rho_2$ sobre las variables de región, obteniendo así un tipo restringido. Este tipo de anotaciones sobre los tipos originales surgen como consecuencia del hecho de que la copia de una estructura de datos debe hacerse en una región distinta de la estructura original.

Esta posibilidad de tener tipos restringidos supone un problema a la hora de decidir de qué forma deben inferirse las restricciones y en qué *momento* de la inferencia debe hacerse esto. Durante la generación de ecuaciones aún no se dispone de información suficientemente concreta para conocer la región externa del tipo que obtenemos al copiar una estructura de datos.

EJEMPLO 7. *Dada la siguiente función que copia la cola de una lista en la región de salida:*

$$\boxed{copyTail\ (x : xs)\ @r = xs@r}$$

Cuando generamos las ecuaciones para la expresión del lado derecho de esta definición partimos del conjunto de suposiciones $\{[copyTail : a_0], [x : a_1], [xs : a_2]\}$*, donde* a_0*,* a_1 *y* a_2 *son variables frescas. La expresión xs@r queda decorada con otra variable fresca* a_3*. Con esta información no podemos imponer la restricción de que la región externa de los tipos xs y su copia sean distintos, ya que estas variables serán sustituidas por los tipos reales tras la resolución del sistema de ecuaciones. Antes de la misma no puede saberse cual es el tipo de la región externa asociada a xs.*

Para tratar este problema, durante el proceso de generación de ecuaciones se generan también un conjunto de restricciones *isData*. Cada restricción de este tipo puede entenderse como una igualdad *latente*, en el sentido de que se procesa después de la resolución del sistema de ecuaciones. Estas restricciones tienen la forma $t = isData(t', \rho)$, donde t y t' serán tipos algebraicos (no básicos) cuando sean sustituidos y ρ es una variable de región. Su significado intuitivo se describe a continuación:

$t = isData(t', \rho)$

> *Los tipos* t *y* t' *son algebraicos, tienen el mismo nombre de constructor y número de parámetros. Ambos unifican en todos sus parámetros, salvo en el de la región más externa de* t *y* t'*, que debe ser distinto. Además, el tipo de la región más externa de* t *ha de ser* ρ*.*

Tras la generación de ecuaciones se tiene un conjunto de restricciones *isData*, cuyas variables serán sustituidas tras la resolución del sistema de ecuaciones generado. El procesado de restricciones tiene como salida una nueva sustitución, un nuevo conjunto de ecuaciones de unificación y un conjunto de desigualdades entre variables de tipo de región. Según la forma de los tipos t y t' que intervengan en una restriccion *isData*, podemos distinguir cuatro casos:

- **Caso 1:** t es un tipo algebraico (no básico) y t' es una variable de tipo:

  $$T\ t_1 \ldots t_n @ \rho_1 \ldots \rho_m = isData(a, \rho)$$

  Sea ρ_{ext} la variable de tipo de región más externa en t y ρ' una variable fresca:

  - Se genera la sustitución $[a \mapsto t[\rho'/\rho_{ext}]]$
  - Se genera la desigualdad $\rho' \neq \rho$
  - Se genera la ecuación $\rho_{ext} = \rho$

- **Caso 2:** t es una variable de tipo y t' es un tipo algebraico (no básico):

  $$a = isData(T\ t_1 \ldots t_n @ \rho_1 \ldots \rho_m, \rho)$$

  Sea ρ_{ext} la variable de tipo región más externa en t':

- Se genera la sustitución $[a \mapsto t'[\rho/\rho_{ext}]]$
- Se genera la desigualdad $\rho_{ext} \neq \rho$

- **Caso 3:** t y t' son ambos tipos algebraicos no básicos:

  $$T\ t_1 \ldots t_n@\rho_1 \ldots \rho_m = isData(T\ t'_1 \ldots t'_n@\rho'_1 \ldots \rho'_m, \rho)$$

  Sean ρ_{ext} y ρ'_{ext} las variables de tipo región más externas de t y t' respectivamente. La restricción supone que las constructoras de tipo de t y t' son ambas iguales y aceptan el mismo número de parámetros n y m. En otro caso tenemos un error de tipos, ya que al copiar una estructura de datos, la copia ha de tener el mismo tipo subyacente[1] que el original, independientemente del hecho de que se sitúe en otra región.

  - Se generan las ecuaciones $t_1 = t'_1, \ldots, t_n = t'_n$.
  - Se generan las ecuaciones $\rho_1 = \rho'_1, \ldots, \rho_m = \rho'_m$ para todas las regiones *no externas* de los tipos t y t'.
  - Se genera la ecuación $\rho_{ext} = \rho$.
  - Se genera la desigualdad $\rho'_{ext} \neq \rho$.

- **Caso 4:** t y t' son ambos variables de tipo:

  $$a = isData(a', \rho)$$

  Este tipo de restricciones surge como consecuencia de que se intenta copiar una estructura cuyo tipo es demasiado general. En particular, no está asegurado que no se esté realizando una copia de un tipo básico. Por tanto, la inferencia de tipos devuelve error.

Una vez resuelta cada restricción por separado, se reunen todas las desigualdades y ecuaciones obtenidas y se realiza la composición de las sustituciones generadas.

3.5. Comprobación de desigualdades

Tras el proceso de resolución de ecuaciones descrito en la Sección 3.3 se obtenía una sustitución θ_1. Tras aplicar esta sustitución al conjunto de restricciones *isData*, se pasaba a procesar dichas restricciones de la forma comentada en la Sección 3.4. Esto producía como resultado una sustitución θ_2 y un nuevo conjunto de ecuaciones a resolver, al cual se aplica el mismo método de resolución de ecuaciones, obteniendo como resultado otra sustitución θ_3. Llamaremos θ a la composición de estas tres sustituciones: $\theta = \theta_3 \circ \theta_2 \circ \theta_1$.

El último paso en la inferencia es la comprobación de las desigualdades acumuladas hasta el momento. Para ello se aplica la sustitución θ a todas las variables de tipo

[1] En este contexto entendemos por tipo subyacente aquel en que se omiten las variables de región.

de región involucrados en tales desigualdades. Una vez hecho esto, basta con comprobar que los nombres de las variables de tipo de región son distintos en cada lado de cada desigualdad. En caso contrario, el usuario intenta copiar una estructura en la misma región en la que habita y, por consiguiente, el algoritmo de inferencia debe devolver error.

Si la comprobación se ha realizado con éxito, se aplica la sustitución θ en cada decoración del árbol abstracto, obteniendo así los tipos definitivos. De este modo, reuniendo el tipo que decora la definición de función junto con las desigualdades provenientes de la resolución de restricciones obtenemos el tipo final de la función.

3.6. Tipo anotado por el usuario

Al igual que en la mayoría de lenguajes funcionales con tipos, el programador puede anotar el tipo de una función antes de la definición de la misma. El tipo especificado por el programador debe cumplir dos condiciones:

- El tipo anotado por el usuario debe ser igual o más *particular* que el tipo obtenido mediante el algoritmo de inferencia. Es decir, si t es el tipo inferido y t' es el tipo anotado, debe existir una sustitución θ sobre las variables de t tal que $\theta(t) = t'$.
- El tipo anotado por el usuario debe ser igual o más *restrictivo* (en cuanto a desigualdades entre variables de tipo de región) que el tipo obtenido mediante el algoritmo de inferencia. Es decir, el conjunto de desigualdades incluidas en el tipo anotado por el usuario debe ser un superconjunto de las desigualdades incluidas en el tipo inferido.

Sea t_u el tipo especificado por el usuario. Para poder realizar la primera comprobación se añade al final del proceso de generación de ecuaciones una igualdad adicional: se iguala t_u con la decoración correspondiente a la definición de la función (esto es, el tipo inferido). Tras el proceso de resolución de ecuaciones y comprobación de restricciones *isData* se obtiene una sustitución final θ que será aplicada a t_u. Tras esto, si t_u y $\theta(t_u)$ son iguales (salvo renombramiento de variables), el tipo especificado por el usuario es igual o más particular que el tipo inferido. En caso contrario, el tipo anotado por el usuario es demasiado general y, por tanto, no es válido.

EJEMPLO 8. *Supongamos que se infiere el siguiente tipo para una función:*

$$a \to b \to Int$$

Mientras que el tipo anotado por el usuario es:

$$t_u = c \to Bool \to Int$$

Tras la resolución de ecuaciones se obtiene la sustitución $\theta \supseteq \{[b \mapsto Bool], [c \mapsto a]\}$*, que aplicada al tipo anotado* t_u *da lugar al tipo* $\theta(t_u) = a \to Bool \to Int$*. Como* $\theta(t_u)$ *es igual que* t_u *(salvo renombramiento de a por c), el tipo especificado por el usuario es más particular que el inferido y, por tanto, es correcto.*

EJEMPLO 9. *Sea t_u el siguiente tipo anotado por el usuario para una función:*

$$t_u = \mathit{Tree\ Int@}\rho_1 \rightarrow \rho_2 \rightarrow \mathit{Tree\ Int@}\rho_2$$

Supongamos que en el tipo inferido se exige que la región del parámetro y la región de salida sean iguales, esto es:

$$\mathit{Tree\ Int@}\rho_3 \rightarrow \rho_3 \rightarrow \mathit{Tree\ Int@}\rho_3$$

Tras la fase de resolución de ecuaciones se obtiene $\theta \supseteq \{[\rho_3 \mapsto \rho_1], [\rho_2 \mapsto \rho_1]\}$. Sustituyendo en t_u se obtiene el tipo Tree Int@$\rho_1 \rightarrow \rho_1 \rightarrow$ Tree Int@ρ_1, que no es igual (módulo renombramiento) al tipo especificado por el usuario. El tipo del usuario es más general que el inferido y, por consiguiente, no es válido.

3.7. Tratamiento especial de la región *self*

El tipo de la región *self* requiere un tratamiento diferente durante la inferencia de tipos. El motivo es que su tipo no puede aparecer en la signatura de ninguna función: la región *self* se destruye cuando la función ha terminado de ser evaluada y, por tanto, ningún parámetro de entrada ni el resultado de la función puede estar total o parcialmente almacenado en la misma.

EJEMPLO 10. *La siguiente definición de función es incorrecta:*

$$\begin{array}{l} \mathit{prefijoCeroCero\ xs\ @r} = \textbf{let}\ \mathit{zs} = (0 : \mathit{xs})\mathit{@self} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (0 : \mathit{zs})\mathit{@r} \end{array}$$

En este caso se está devolviendo una lista cuya cola se encuentra contenida en self.

Durante el proceso de generación de ecuaciones se asociaban variables de tipo frescas a las variables de región que iban siendo recorridas (incluyendo *self*). Introduciremos un mecanismo que permita distinguir si una variable fresca fue creada para el tipo de una variable de región distinta de *self* o fue creada para el tipo de *self*. En este último caso diremos que dicha variable de tipo de región está *asociada a self*.

La comprobación para asegurar que no se devuelve ninguna estructura contenida en *self* se limita ahora a comprobar que en el tipo de la función que se está definiendo no aparece ninguna variable asociada a *self*. El mecanismo que se utilizará para distinguir entre variables asociadas y no asociadas a *self* es bastante sencillo. Las primeras tendrán nombres de la forma tself_0, tself_1,... mientras que el resto serán de la forma ρ_0, ρ_1, ...

El uso de este mecanismo necesita de cierta cautela durante la generación de ecuaciones. En el ejemplo anterior la región externa de *zs* tiene tipo tself_0, mientras que el resultado se guarda en la región r, de tipo ρ_0. Estas dos variables de tipo acabarán siendo unificadas. Si la sustitución generada contuviese la ligadura $[\mathit{tself}_0 \mapsto \rho_0]$ todas las apariciones de tself_0 contenidas en el tipo de la función *prefijoCeroCero* serían sustituidas por una variable no asociada a *self*, con lo que no se detectaría que la función es

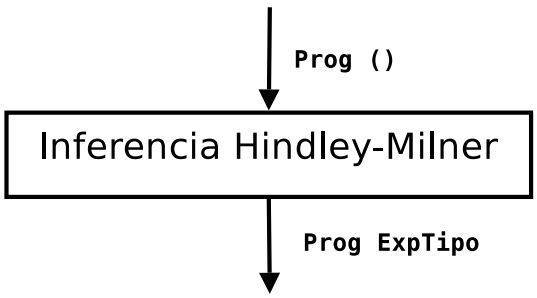

Figura 3.6: Entrada y salida de la inferencia de tipos Hindley-Milner

incorrecta. Si la sustitución generada hubiese contenido la ligadura $[\rho_0 \mapsto tself_0]$, sí se hubiera detectado.

Para evitar sustituciones que reemplacen una variable asociada a *self* por una no asociada se evitará generar ecuaciones que tengan sólo en el lado izquierdo una variable asociada a *self*. Si se intenta generar una ecuación de esta forma, se intercambiarán ambos lados de la ecuación. De esta forma quedará garantizado que no se generen ligaduras que hagan "desaparecer" regiones asociadas a *self* de los tipos.

3.8. Detalles de implementación

Una vez explicados todos los conceptos y procedimientos que tienen lugar en esta fase de compilación, se comentarán a continuación aquellos aspectos relativos a la implementación de la misma. Tal como se muestra en la Figura 3.6, este análisis parte del árbol abstracto correspondiente a un programa, cuyas decoraciones son todas igual al valor *Unit* de Haskell (cuyo tipo viene representado por el símbolo( )) y obtiene como resultado el mismo árbol abstracto, donde cada expresión y definición está decorada con su tipo.

Antes de exponer los detalles de la implementación de cada procedimiento por separado conviene explicar cómo se relacionan todos estos procedimientos explicados en las secciones anteriores con el fin de tener una visión global del proceso de inferencia de tipos. En la Figura 3.7 se muestra un esquema de todos los elementos generados que participan en dicha inferencia.

El proceso es el siguiente: recibimos como entrada el árbol abstracto correspondiente al programa. Mediante el proceso de generación de ecuaciones expuesto en la Sección 3.2 se obtiene un árbol abstracto decorado con tipos. En este punto la mayoría de los tipos que sirven como decoración son variables de tipo, las cuales serán posteriormente sustituidas por los tipos finales. Por otro lado, se devuelve un conjunto de ecuaciones y un conjunto de restricciones *isData*. Las primeras se resuelven para obtener una sustitución θ_1, que será aplicada a las restricciones *isData* que se han obtenido. La resolución de restricciones, por su parte, generará una sustitución θ_2, un conjunto de ecuaciones (que serán resueltas para obtener θ_3) y una lista de desigualdades. La

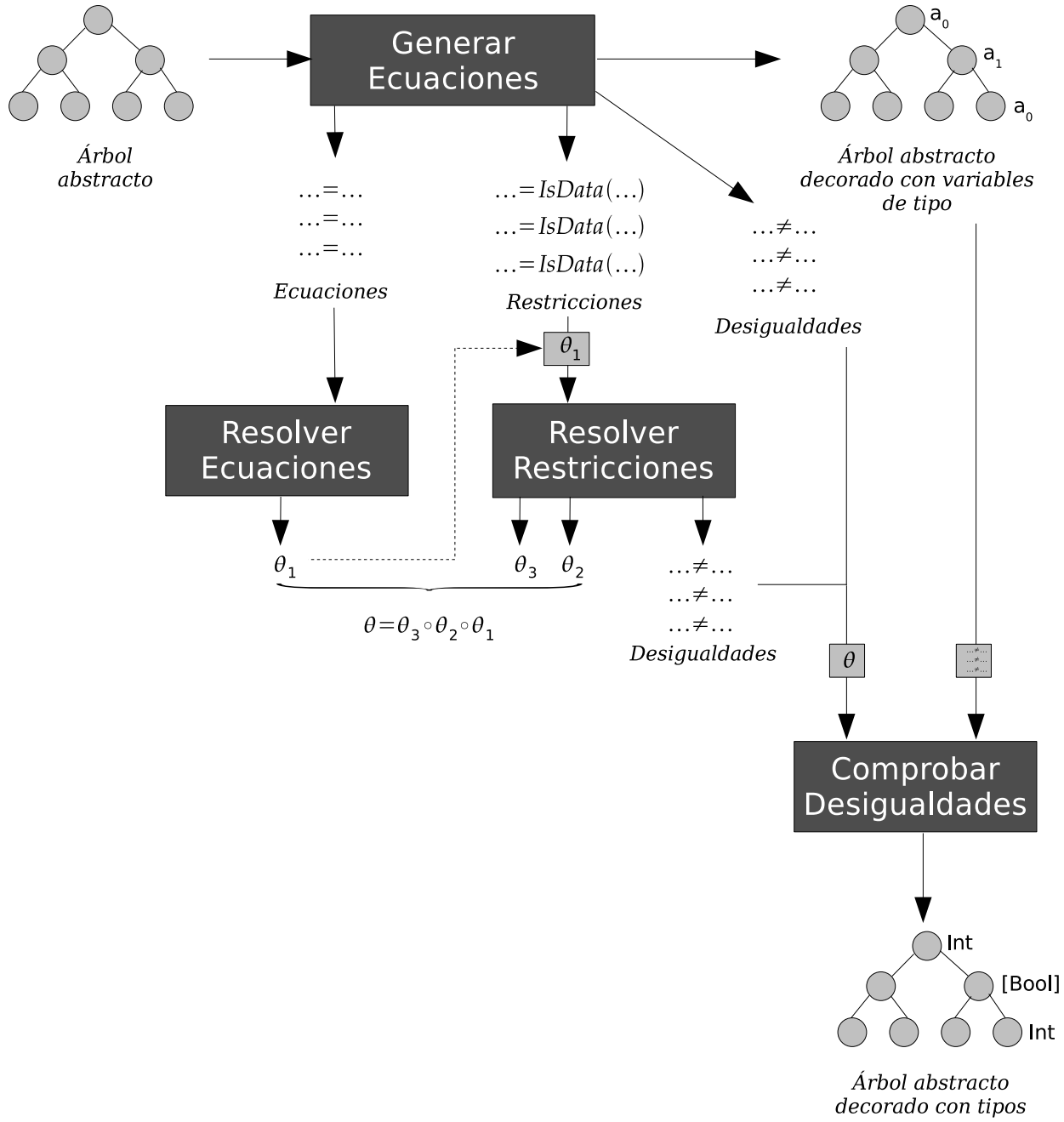

Figura 3.7: Fases y elementos generados durante la inferencia de tipos Hindley-Milner

composición de θ_3, θ_2 y θ_1 es aplicada al árbol abstracto decorado con variables para obtener los tipos definitivos. Además se aplica al conjunto de desigualdades generado, para comprobar que las variables de tipo de región a ambos lados de una desigualdad son realmente distintas.

3.8.1. Estado de generación de ecuaciones

Para implementar la fase de generación de ecuaciones se parte de las reglas representadas en las Figuras 3.2, 3.3 y 3.4. En estas reglas es necesaria a menudo la generación de un nombre fresco de variable. La generación de nombres frescos de variables es particularmente laboriosa de implementar en un lenguaje funcional como Haskell, ya que al no disponer de estado es necesario propagar a lo largo del recorrido del árbol abstracto información actualizable que sirva como punto de partida para generar nombres de variable frescos. Además de esta información es necesario acumular las ecuaciones, restricciones *isData* y desigualdades producidas por cada regla. Estas cuatro componentes formarán parte de un *estado* que se propagará explícitamente por el árbol abstracto. El estado queda definido por una tupla de cuatro componentes:

```
type State = ([Equation], [Constraint], [Inequality], Int)
```

La primera componente del estado es una lista que acumula las ecuaciones generadas. El tipo de datos `Equation` se define como un par ordenado de tipos, aunque para construir nuevas ecuaciones se utilizará el operador infijo `(===)`, que construye la tupla atendiendo a dos restricciones:

1. Si uno de los miembros de la ecuación es una variable de tipo no asociada a la región *self*, ésta será la primera componente de la tupla.

2. Si uno de los miembros de la ecuación es una variable de tipo asociada a la región *self*, deberá ser la segunda componente de la tupla.

La primera restricción resulta de utilidad para reducir los casos posibles durante el proceso de resolución de ecuaciones. La segunda restricción evita que las variables de tipo asociadas a *self* sean sustituidas. Para generar una ecuación se dispone de la función `addEquation`, con la siguiente signatura:

```
addEquation :: Equation -> State -> State
```

La segunda componente del estado es una lista que acumula las restricciones *isData* acumuladas. El tipo de datos `Constraint` está definido del siguiente modo:

```
data Constraint = IsData ExpTipo ExpTipo VarTipo
```

Donde un término Haskell de la forma `IsData t t' rho` representa a la restricción $t = isData(t', rho)$. Para acumular una nueva restricción se dispone de la función `addConstraint`.

```
addConstraint :: Constraint -> State -> State
```

La tercera componente del estado contiene las desigualdades entre variables de tipo de región que van generándose durante el recorrido del árbol abstracto. Mediante la función `addInequality` puede añadirse una nueva desigualdad:

```
data Inequality = NotEqual VarTipo VarTipo

addInequality :: Inequality -> State -> State
```

Para mantener la definición del estado encapsulada, disponemos de tres funciones que acceden a estas tres componentes:

```
equations    :: State -> [Equation]
constraints  :: State -> [Constraint]
inequalities :: State -> [Inequality]
```

Las variables frescas de tipo generadas son de la forma `a0`, `a1`, `a2` ... para variables no asociadas a *self* e identificadores `tself0`, `tself1`, `tself2` ... para variables asociadas a *self*. El número que aparece como sufijo de cada nombre es suficiente para asegurar que las variables que se generan no han sido utilizadas anteriormente. Para ello, incluimos como cuarta componente en el estado un entero que indique el número que corresponde a la última variable fresca generada. Si desde el proceso de generación de ecuaciones necesitamos un nombre fresco de variable, utilizaremos las funciones `freshVar` y `freshSVar`:

```
freshVar :: State -> (State, VarTipo)
freshVar (eqs,cons,ineq,n) = ((eqs, cons, ineq, n+1), "a" ++ show n)

freshSVar :: State -> (State, VarTipo)
freshSVar (eqs,cons,ineq,n) = ((eqs, cons, ineq, n+1), "tself" ++ show n)
```

3.8.2. Generación de ecuaciones

La función `decorAndGenExp` se encarga del primer recorrido del árbol abstracto con el fin de generar las ecuaciones, restricciones y desigualdades correspondientes a cada expresión del programa. El tipo de esta función es:

```
decorAndGenExp :: Maybe String -> Assumps -> State -> Exp a
               -> (State, Exp ExpTipo)
```

Donde `Assumps` se encuentra definido del siguiente modo:

```
type Assumps = M.Map String ExpTipo
```

El tipo `Map` se encuentra definido en el módulo `Data.Map` de las librerías de GHC [Ada93] e implementa una tabla como un árbol binario equilibrado.

El primer parámetro de `decorAndGenExp` contiene el nombre de la función cuya definición está siendo procesada. Si se está procesando la expresión principal del programa este parámetro tendrá el valor `Nothing`. El segundo parámetro contiene el tipo de cada identificador que actualmente está en ámbito. Se corresponde con el conjunto de suposiciones A de las reglas de la Sección 3.2. El tercer parámetro es el estado que se propaga a lo largo del recorrido del árbol abstracto y cuyas componentes han sido explicadas en la sección anterior. El tercer parámetro es la expresión de entrada. El resultado contiene el estado actualizado con las ecuaciones, restricciones *isData* y desigualdades generadas incluídas en el mismo y, en su caso, con el contador de variables frescas actualizado. También se devuelve el árbol abstracto de la expresión con la decoración añadida.

La definición de esta función es una traducción directa de las reglas contenidas en la Figura 3.2. A modo de ejemplo se muestra la regla correspondiente a una expresión **case** y la definición de `decorAndGenExp` encargada de procesarla.

$$\frac{A \vdash e : t \mid E \quad \forall i \left(A \vdash_p p_i : t_i \mid A_i, E_i \;\wedge\; A \cup A_i \vdash e : t_i' \mid E_i', C_i', I_i'\right)}{\begin{array}{rl} A \vdash \mathbf{case}(!)\ e\ \mathbf{of}\ \overline{p_i \rightarrow e_i}^n : t_1 \mid & (\bigcup_i E_i) \cup (\bigcup_i E_i') \cup E \cup E_x, \\ & (\bigcup_i C_i) \cup (\bigcup_i C_i') \cup C, \\ & (\bigcup_i I_i) \cup (\bigcup_i I_i') \cup I \end{array}}$$

$$\textbf{donde } E_x = \{\ \begin{array}{llll} t = t_1, & t_1 = t_2, & \ldots, & t_{n-1} = t_n, \\ & t_1' = t_2', & \ldots & t_{n-1}' = t_n' \end{array}\ \}$$

```
decorAndGenExp ms as st (CaseE exp alts _) =
                                        (st', CaseE exp' alts' (head ts'))
  where (st1,exp') = decorAndGenExp ms as st exp
        (st2,alts') = L.mapAccumL (decorAndGenAltCase ms as) st1 alts
        (ts, ts') = (unzip . map (\(p,e) -> (patType p, decExp e))) alts'
        st3 = addEquation (decExp exp' === head ts) st2
        eqs1 = zipWith (===) ts (tail ts)
        eqs2 = zipWith (===) ts' (tail ts')
        st' = foldr addEquation st3 (eqs1 ++ eqs2)
```

En primer lugar se procesa el discriminante `exp`, dando lugar al estado `st1`. Este estado sirve de partida para procesar cada alternativa, obteniendo como resultado el estado `st2`. Al añadir la ecuación $t = t_1$ se obtiene el estado `st3`, al cual se le añade el resto de ecuaciones contenidas en E_x, siendo `st'` el estado resultante.

La función `decorAndGenPat` se encarga de implementar las reglas contenidas en la Figura 3.3. Su signatura es análoga a la de `decorAndGenExp`, con la diferencia de que se omite el primer parámetro (al resultar innecesario) y se devuelve la tabla de suposiciones con la información correspondiente a las variables del patrón introducida:

```
decorAndGenPat :: Assumps -> State -> Patron a
               -> (Assumps, State, Patron ExpTipo)
```

En un programa *Full-Safe* una función puede quedar definida mediante múltiples ecuaciones. La función `decorAndGenOuterDef` se encarga de procesar cada ecuación y su implementación se corresponde con la última regla de la Figura 3.4:

```
decorAndGenOuterDef :: Assumps -> State -> Def a -> (State, Def ExpTipo)
decorAndGenOuterDef as st (t, (nom, patBools, []), der) =
                (st5,(t ++ [VarT fv], (nom, zip pats' bools, []), der'))
 where (pats, bools) = unzip patBools
       (st1, fv) = freshVar st
       (as', st2, pats') = decorAndGenPats as st1 pats
       (st3, fvSelf) = freshSVar st2
       (st4, der') = decorAndGenDer (Just nom)
           (M.insert "self" (VarT fvSelf) (M.insert nom (VarT fv) as'))
                                                          st3 der
       st5 = addEquation
               (VarT fv === foldr Flecha (derType der') (map patType pats'))
               st4
```

3.8.3. Sustituciones y resolución de ecuaciones

El siguiente paso en la inferencia de tipos es la resolución del sistema de ecuaciones entre tipos generado. El resultado de la misma es una sustitución, que es implementada como una tabla que asocia nombres de variables con tipos.

```
type Subst = M.Map VarTipo ExpTipo
```

Junto con el tipo de datos `Subst` se definen la sustitución vacía (`emptySubst`), la sustitución unitaria (`unitSubst`) y la composición de sustituciones (`<.>`):

```
infixr 6 <.>
emptySubst :: Subst
unitSubst  :: VarTipo -> ExpTipo -> Subst
(<.>)      :: Subst -> Subst -> Subst
```

Para aplicar una sustitución a una determinada expresión de tipo se utiliza la función `substitute`:

```
substitute :: Subst -> ExpTipo -> ExpTipo
```

La implementación de la resolución de ecuaciones resulta bastante sencilla; las reglas de la Figura 3.5 se traducen en la función `solveEqs'`, que además de la lista de ecuaciones a resolver recibe un parámetro acumulador que contiene la sustitución construida hasta el momento.

```
solveEqs' :: [Equation] -> Subst -> Maybe Subst
```

Si el sistema de ecuaciones no tiene solución se devuelve el valor `Nothing`. La implementación de esta función sigue las reglas de la Figura 3.5. A continuación se muestra la implementación para las ecuaciones de la forma $a = t$:

$$\{a = t\} \uplus E, \theta \implies E\,[t/a]\,, [a \mapsto t] \circ \theta \quad \text{si } a \notin FV(t)$$

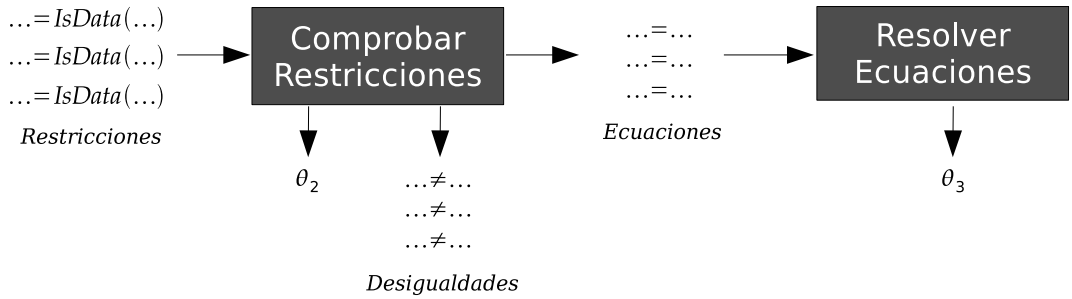

Figura 3.8: Proceso de resolución de restricciones *isData*

```
solveEqs' ((VarT a,t):eqs) s | not (varInTypeExp a t) =
               let newSubst = unitSubst a t
                in solveEqs' (applySubst newSubst eqs) (newSubst <.> s)
```

El proceso de resolución de ecuaciones comienza con la sustitución vacía. La función `solveEqs` realiza la primera llamada a `solveEqs'` con dicha sustitución:

```
solveEqs :: [Equation] -> Maybe Subst
solveEqs eqs = solveEqs' eqs emptySubst
```

3.8.4. Comprobación de restricciones y desigualdades

La sustitución θ_1 obtenida tras el proceso de resolución de ecuaciones se aplica al conjunto de restricciones *isData*, tal como se mostraba en la Figura 3.7. A continuación se realizarán comprobaciones sobre las restricciones siguiendo los casos expuestos en la Sección 3.4. Estas comprobaciones, en caso de tener éxito (esto es, no se da el Caso 4) dan lugar a:

- Una sustitución θ_2.
- Un conjunto de ecuaciones.
- Un conjunto de desigualdades.

Cada restricción por separado puede generar ninguno, uno o varios de estos elementos. Las desigualdades se añaden a la estructura `State`, mientras que las ecuaciones y sustituciones se van acumulando. A continuación se muestra la signatura de la función `checkConstraint`, encargada de la comprobación de una restricción *isData*.

```
checkConstraint :: State -> Constraint
                -> (Maybe [Equation], Maybe Subst, State)
```

Por último, la función `checkConstraints` se encarga de reunir todas las ecuaciones y todas las sustituciones provenientes de analizar cada restricción por separado y utiliza el resolutor de ecuaciones para resolver las mismas, obteniendo así θ_3 (Figura 3.8).

```
checkConstraints :: State -> [Constraint] -> (Maybe Subst, State)
```

El resultado de la función es la composición de θ_3 con θ_2 si el proceso de resolución de ecuaciones tuvo éxito, o `Nothing` en caso contrario. Adicionalmente se devuelve el estado modificado con las nuevas inecuaciones añadidas.

El último paso en la inferencia de tipos es la aplicación de la sustitución compuesta a partir de los resultados de `checkConstraints` y `solveEqs` al conjunto de desigualdades y la comprobación de que a ambos lados de cada desigualdad las variables son diferentes. De ello se encarga la función `checkInequalities`, que recibe una lista de desigualdades, y devuelve la primera desigualdad que no cumple este requisito, en el caso de existir alguna. Si en todas las desigualdades se cumple que las variables son distintas, el resultado es `Nothing`:

```
checkInequalities :: [Inequality] -> Maybe Inequality
```

3.9. Ejemplos

En esta última sección se detallará el funcionamiento del algoritmo de tipos mediante varios ejemplos. En primer lugar se mostraremos cómo se infiere el tipo de una función sencilla que calcula la longitud de una lista.

EJEMPLO 11. *Se parte de la siguiente función length, que devuelve la longitud de una lista:*

$$\begin{array}{|l} \hline \mathit{length}\ [\,] = 0 \\ \mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs \\ \hline \end{array}$$

Por otro lado, se suponen ya conocidas las signaturas de las constructuras de lista vacía $([\,])$*, lista no vacía* $(:)$ *y el operador suma* $(+)$*:*

$$\begin{array}{rcl} [\,] & :: & \rho \to [a]@\rho \\ (:) & :: & a \to [a]@\rho \to \rho \to [a]@\rho \\ (+) & :: & \mathit{Int} \to \mathit{Int} \to \mathit{Int} \end{array}$$

Se comienza asignando una variable fresca a_0 *como tipo de length. A continuación se examina cada ecuación de esta definición por separado.*

Ecuación 1. *Se decora el patrón* $[\,]$ *con una variable fresca* a_1*. Del mismo modo se obtiene una instancia fresca de la constructora* $[\,]$*, esto es,* $\rho_0 \to [a_2]@\rho_0$*. La primera ecuación generada es:*

$$(\mathbf{E1}) \qquad \rho_0 \to [a_2]@\rho_0 = \rho_1 \to a_1$$

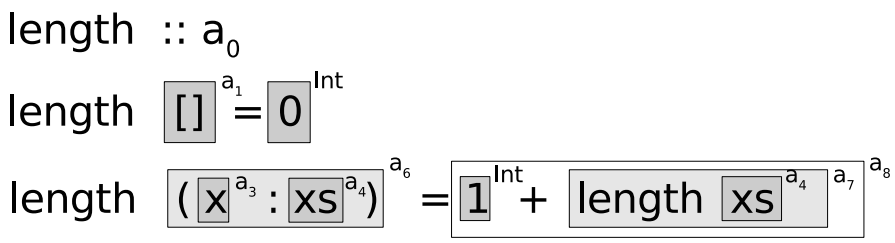

Figura 3.9: Decoración de la definición tras generar las ecuaciones

donde ρ_1 es una variable fresca. Para la decoración del lado derecho de la ecuación se parte del conjunto de suposiciones $A = \{[self : \rho_2], [length : a_0]\}$, que no son necesarias, pues en el lado derecho sólo tenemos una constante numérica, que se decora con el tipo Int. La función, por tanto, recibe un parámetro de tipo a_1 y devuelve un valor de tipo Int, lo cual genera la siguiente ecuación:

$$(\mathbf{E2}) \qquad a_0 = a_1 \rightarrow Int$$

Ecuación 2. *Se comienza decorando el patrón y cada uno de sus subpatrones con variables frescas, obteniendo $x : a_3$, $xs : a_4$ y $(x : xs) : a_6$. Tras la obtención de una instancia fresca para el constructor (:) ($a_5 \rightarrow [a_5]@\rho_3 \rightarrow \rho_3 \rightarrow [a_5]@\rho_3$) se genera la siguiente ecuación:*

$$(\mathbf{E3}) \qquad a_5 \rightarrow [a_5]@\rho_3 \rightarrow \rho_3 \rightarrow [a_5]@\rho_3 = a_3 \rightarrow a_4 \rightarrow \rho_4 \rightarrow a_6$$

Una vez decorado el patrón del lado izquierdo, puede formarse el conjunto de suposiciones $A = \{[x : a_3], [xs : a_4], [self : \rho_5], [length : a_0]\}$, con el que se procederá a decorar el lado derecho. En el mismo se encuentra la función suma aplicada a dos argumentos. El primero de ellos es decorado con el tipo Int, mientras que el segundo es una llamada recursiva a la función length, que será decorada con una variable fresca a_7 e igualando con el tipo de dicha función (que es a_0), se genera la siguiente ecuación:

$$(\mathbf{E4}) \qquad a_0 = a_4 \rightarrow a_7$$

Donde a_4 es el tipo de la variable xs, obtenido a partir del conjunto de suposiciones. Del mismo modo se decora la llamada a la función $(+)$ con la variable fresca a_8 y se genera otra ecuación:

$$(\mathbf{E5}) \qquad Int \rightarrow Int \rightarrow Int = Int \rightarrow a_7 \rightarrow a_8$$

El lado derecho de la definición queda, por tanto, decorada con tipo a_8 y su primer parámetro quedaba decorado con a_6. A partir de esta información podemos generar la última ecuación:

$$(\mathbf{E6}) \qquad a_0 = a_6 \rightarrow a_8$$

Cada elemento del árbol abstracto queda decorado del modo indicado en la Figura 3.9. El siguiente paso es la resolución del sistema de ecuaciones planteado utilizando las reglas de la

Figura 3.5. Mostraremos la evolución del conjunto E de ecuaciones y la sustitución θ generada mediante el uso de estas reglas. Por motivos de espacio las ecuaciones que no son relevantes en el paso actual serán mostradas con sus nombres $(E1)\ldots(E6)$ y mediante el símbolo $\Rightarrow^*$ indicaremos la aplicación sucesiva de varias reglas.

$$
\begin{array}{rcl}
E & \equiv & \{\rho_0 \to [a_2]@\rho_0 = \rho_1 \to a_1, (E2), (E3), (E4), (E5), (E6)\} \\
\theta & \equiv & \emptyset \\
\Rightarrow & & \\
E & \equiv & \{\rho_0 = \rho_1, a_1 = [a_2]@\rho_0, (E2), (E3), (E4), (E5), (E6)\} \\
\theta & \equiv & \emptyset \\
\Rightarrow & & \\
E & \equiv & \{a_1 = [a_2]@\rho_0, \theta(E2), \theta(E3), \theta(E4), \theta(E5), \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_1]\} \\
\Rightarrow & & \\
E & \equiv & \{a_0 = [a_2]@\rho_1 \to Int, \theta(E3), \theta(E4), \theta(E5), \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_1], [a_1 \mapsto [a_2]@\rho_1]\} \\
\Rightarrow & & \\
E & \equiv & \{a_5 \to [a_5]@\rho_3 \to \rho_3 \to [a_5]@\rho_3 = a_3 \to a_4 \to \rho_4 \to a_6, \\
 & & \theta(E4), \theta(E5), \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_1], [a_1 \mapsto [a_2]@\rho_1], [a_0 \mapsto ([a_2]@\rho_1 \to Int)]\} \\
\Rightarrow^* & & \\
E & \equiv & \{[a_2]@\rho_1 \to Int = [a_3]@\rho_4 \to a_7,\ \theta(E5), \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_1], [a_1 \mapsto [a_2]@\rho_1], [a_0 \mapsto ([a_2]@\rho_1 \to Int)] \\
 & & [a_5 \mapsto a_3], [a_4 \mapsto [a_3]@\rho_4], [\rho_3 \mapsto \rho_4], [a_6 \mapsto [a_3]@\rho_4]\} \\
\Rightarrow^* & & \\
E & \equiv & \{a_2 = a_3, \rho_1 = \rho_4, a_7 = Int,\ \theta(E5), \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_1], [a_1 \mapsto [a_2]@\rho_1], [a_0 \mapsto ([a_2]@\rho_1 \to Int)] \\
 & & [a_5 \mapsto a_3], [a_4 \mapsto [a_3]@\rho_4], [\rho_3 \mapsto \rho_4], [a_6 \mapsto [a_3]@\rho_4]\} \\
\Rightarrow^* & & \\
E & \equiv & \{Int \to Int \to Int = Int \to Int \to a_8,\ \theta(E6)\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_4], [a_1 \mapsto [a_3]@\rho_4], [a_0 \mapsto ([a_3]@\rho_4 \to Int)] \\
 & & [a_5 \mapsto a_3], [a_4 \mapsto [a_3]@\rho_4], [\rho_3 \mapsto \rho_4], [a_6 \mapsto [a_3]@\rho_4] \\
 & & [a_2 \mapsto a_3], [\rho_1 \mapsto \rho_4], [a_7 \mapsto Int]\} \\
\Rightarrow^* & & \\
E & \equiv & \{[a_3]@\rho_4 \to Int = [a_3]@\rho_4 \to Int\} \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_4], [a_1 \mapsto [a_3]@\rho_4], [a_0 \mapsto ([a_3]@\rho_4 \to Int)] \\
 & & [a_5 \mapsto a_3], [a_4 \mapsto [a_3]@\rho_4], [\rho_3 \mapsto \rho_4], [a_6 \mapsto [a_3]@\rho_4] \\
 & & [a_2 \mapsto a_3], [\rho_1 \mapsto \rho_4], [a_7 \mapsto Int], [a_8 \mapsto Int]\} \\
\Rightarrow & & \\
E & \equiv & \emptyset \\
\theta & \equiv & \{[\rho_0 \mapsto \rho_4], [a_1 \mapsto [a_3]@\rho_4], [a_0 \mapsto ([a_3]@\rho_4 \to Int)] \\
 & & [a_5 \mapsto a_3], [a_4 \mapsto [a_3]@\rho_4], [\rho_3 \mapsto \rho_4], [a_6 \mapsto [a_3]@\rho_4] \\
 & & [a_2 \mapsto a_3], [\rho_1 \mapsto \rho_4], [a_7 \mapsto Int], [a_8 \mapsto Int]\}
\end{array}
$$

Como no se ha generado ninguna restricción isData ni ninguna desigualdad que proce-

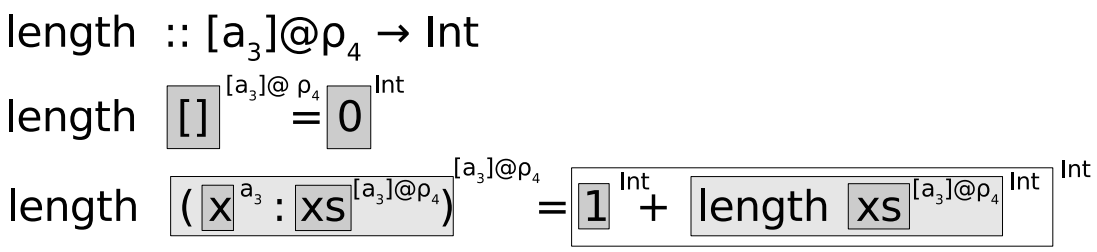

Figura 3.10: Decoración de la definición tras aplicar la sustitución θ

sar, puede aplicarse directamente la sustitución θ al árbol abstracto, obteniendo los tipos de la Figura 3.10. El tipo inferido para la función length es:

$$length \ :: \ [a_3]@\rho_4 \rightarrow Int$$

A continuación se mostrará el funcionamiento de las restricciones *isData* y las desigualdades generadas por las mismas en una sencilla función *copia*:

EJEMPLO 12. *Sea la siguiente definición que, partiendo de una estructura y una región destino, crea una nueva copia de la estructura en dicha región:*

$$copia\ xs\ @r = xs@r$$

El proceso de generación de ecuaciones resulta bastante sencillo. Se decora la función copia y su parámetro con las variables frescas a_0 y a_1, respectivamente. Para la variable de región se necesita otra variable fresca ρ_0. Con el conjunto de suposiciones $A = \{[copia : a_0], [xs : a_1], [r : \rho_0], [self : tself_1]\}$ podemos pasar a generar las ecuaciones para el lado derecho de la definición. La expresión xs@r pasa a ser decorada con otra variable fresca a_2. En total se genera una restricción isData y una ecuación:

$$(\mathbf{I1}) \quad a_2 = isData(a_1, \rho_0)$$
$$(\mathbf{E1}) \quad a_0 = a_1 \rightarrow \rho_0 \rightarrow a_2$$

Tras resolver la única ecuación se obtiene la sustitución $\theta = \{a_0 \mapsto (a_1 \rightarrow \rho_0 \rightarrow a_2)\}$, que no tiene ningún efecto en la restricción $(I1)$. Si se intenta resolver dicha restricción con los casos contemplados en la Sección 3.4 se verá que sólo es aplicable el Caso 4, con lo que la resolución no tiene éxito.

En este ejemplo se ha intentado mostrar una función demasiado general. Se intenta hacer una copia de un dato de tipo a_1. En particular podría utilizarse esta función para copiar una variable que contuviese un tipo básico (entero o booleano), lo cual no está permitido, ya que sólo pueden hacer copias de estructuras de datos. Por tanto, no es posible en *Full-Safe* realizar una función genérica de copia aplicable a todos los tipos de datos. Si se quiere restringir a un determinado tipo de datos (como por ejemplo listas) puede incluirse una anotación de tipo en la función, como se verá en el siguiente ejemplo:

EJEMPLO 13. *Se opta por restringir esta función de copia al tipo de datos lista, incluyendo una anotación de tipo:*

$$\begin{array}{|l|}\hline copia \;::\; [a]@\rho \to \rho' \to [a]@\rho' \\ copia\ xs\ @r = xs@r \\ \hline\end{array}$$

El proceso de generación de ecuaciones es idéntico al del ejemplo anterior, con la diferencia que se añade una nueva ecuación que iguala el tipo inferido con el tipo anotado por el usuario. El resultado es:

$$\begin{array}{ll}(\mathbf{I1}) & a_2 = isData(a_1, \rho_0) \\ (\mathbf{E1}) & a_0 = a_1 \to \rho_0 \to a_2 \\ (\mathbf{E2}) & a_0 = [a]@\rho \to \rho' \to [a]@\rho'\end{array}$$

La resolución de las ecuaciones $(E1)$ *y* $(E2)$ *da lugar a la sustitucion:*

$$\theta = \{[a_0 \mapsto ([a]@\rho \to \rho' \to [a]@\rho')], [a_1 \mapsto [a]@\rho], [\rho_0 \mapsto \rho'], [a_2 \mapsto [a]@\rho']\}$$

que aplicada a la restricción isData da lugar a:

$$[a]@\rho' = isData([a]@\rho, \rho')$$

Al resolver esta restricción se obtienen dos ecuaciones de resolución trivial, $\rho' \neq \rho'$ *y* $a = a$*, junto con la desigualdad* $\rho = \rho'$*. El tipo inferido para la función copia es:*

$$copia \;::\; \rho \neq \rho' \;.\; [a]@\rho \to \rho' \to [a]@\rho'$$

El algoritmo de inferencia de tipos devolverá error en el tipo especificado por el usuario, ya que es menos restrictivo que el tipo inferido. Incluyendo esta restricción en la definición se obtiene la versión definitiva de la función copia para listas:

$$\begin{array}{|l|}\hline copia \;::\; \rho \neq \rho' \;.\; [a]@\rho \to \rho' \to [a]@\rho' \\ copia\ xs\ @r = xs@r \\ \hline\end{array}$$

que finalmente es aceptada por el algoritmo.

Capítulo 4

Análisis de compartición

En el Capítulo 2 se explicó que dadas dos estructuras de datos, una puede ser parte de la otra, o bien pueden compartir una tercera. Antes de comprobar si las destrucciones de estructuras de datos (mediante **case**!) se realizan de forma segura debemos analizar si se produce compartición entre las variables del programa y determinar, en su caso, de qué tipo de compartición se trata.

En este capítulo definiremos un análisis que nos permita aproximar las relaciones de compartición entre dichas variables. Este análisis fue publicado en [PSM06]. En este capítulo haremos un resumen de lo expuesto en esta publicación.

4.1. Tipos de relaciones de compartición

Para poder formalizar las distintas formas en las que puede producirse compartición entre dos estructuras de datos disponemos de cuatro relaciones binarias entre las variables en ámbito del programa (Figura 4.1).

DEFINICIÓN 4. *Dadas dos variables en ámbito x e y:*

$$\begin{array}{rcl} x \lhd\!\!\sim y & \Leftrightarrow_{def} & x \text{ es un descendiente recursivo de } y \\ x \triangle\!\!\sim y & \Leftrightarrow_{def} & x \text{ comparte un descendiente recursivo de } y \\ x \lhd y & \Leftrightarrow_{def} & x \text{ es un descendiente de } y \\ x \triangle y & \Leftrightarrow_{def} & x \text{ comparte un descendiente de } y \end{array}$$

Cabe destacar que las cuatro relaciones son reflexivas, $\triangle$ es simétrica y $\lhd\!\!\sim$ y $\lhd$ son transitivas, lo cual deberá tenerse en cuenta a la hora de implementar las estructuras de datos utilizadas en el análisis. Por otro lado, tenemos las siguientes implicaciones entre las relaciones:

$$\begin{array}{ccccccc} x \lhd\!\!\sim y & \Rightarrow & x \triangle\!\!\sim y & \Rightarrow & x \triangle y \\ x \lhd\!\!\sim y & \Rightarrow & x \lhd y & \Rightarrow & x \triangle y \end{array}$$

El análisis consiste en el recorrido descendente del árbol abstracto, mientras se van acumulando estas relaciones a medida que una variable ligada pasa a estar en ámbito (lo cual ocurre en los recorridos de expresiones **let**, **case** y **case**!)

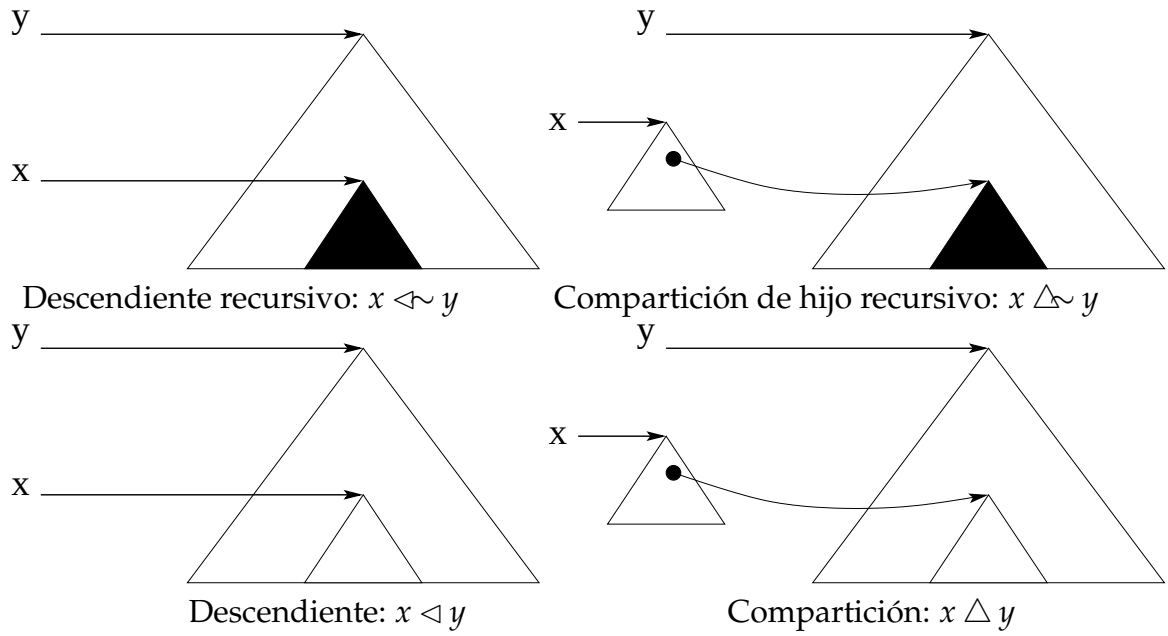

Figura 4.1: Relaciones de compartición entre distintas variables. Los subárboles sombreados representan descendientes recursivos.

4.2. Análisis de las expresiones

Una vez definidas estas cuatro relaciones de compartición, pasamos a describir el análisis de dichas relaciones entre variables y expresiones del programa. Para ello definiremos una interpretación abstracta S que devuelve siete conjuntos para cada expresión e. La idea intuitiva de esta interpretación es la siguiente: Si e se ejecuta y da lugar a una estructura de datos, S estimará por exceso las relaciones de compartición de dicha estructura con las variables en ámbito de e. El significado concreto de los siete conjuntos devueltos se indica a continuación:

1. El conjunto *SubRP* de todas las variables x en ámbito de e tales que $e \lessdot\!\!\sim x$.
2. El conjunto *ShRP* de todas las variables x en ámbito de e tales que $e \triangle\!\!\sim x$.
3. El conjunto *SubP* de todas las variables x en ámbito de e tales que $e \lhd x$.
4. El conjunto *SubR* de todas las variables x en ámbito de e tales que $x \lessdot\!\!\sim e$.
5. El conjunto *ShR* de todas las variables x en ámbito de e tales que $x \triangle\!\!\sim e$.
6. El conjunto *Sub* de todas las variables x en ámbito de e tales que $x \lhd e$.
7. El conjunto *Sh* de todas las variables x en ámbito de e tales que $x \triangle e$.

La interpretación S para las expresiones mantiene cuatro parámetros en los que se acumularán las cuatro relaciones de compartición entre variables. Además recibe un entorno ρ del cual podemos obtener la signatura de las funciones definidas anteriormente. Dicha signatura consiste en siete conjuntos de enteros, cuyo significado es el mismo que el comentado anteriormente con la diferencia de que aquí sólo se almacenan los índices correspondientes a los parámetros de la función.

Por motivos de claridad, introducimos la siguiente notación en la definición de la interpretación S: las relaciones no simétricas ($\lessdot\!\!\sim$, $\lessdot$ y $\triangle\!\!\!\sim$) pueden tratarse como funciones $Var \rightarrow \mathcal{P}(Var)$, llamadas *SubR*, *Sub* y *ShR*, respectivamente. De este modo denominaremos $R(x)$ al conjunto de variables y que cumplen $(y, x) \in R$. También escribiremos $R = [x \rightarrow S]$ para indicar que $S = R(x)$, para cierta variable x.

Por otro lado, la relación simétrica $\triangle$ será tratada como un conjunto de conjuntos, de modo que si tenemos un conjunto $S \in \triangle$, entonces $x \triangle y$ para todas las variables x e y contenidas en S. Por motivos de uniformidad y aún cuando la relación $\triangle$ no se trata de una función, utilizaremos la notación $Sh(x)$ para referirnos al conjunto de variables que comparten un hijo de x, es decir:

$$Sh(x) = \bigcup \{S \mid x \in S \wedge S \in \triangle\}$$

En la Figura 4.2 podemos encontrar la definición completa de la interpretación abstracta S. En esta definición se distinguen casos según la forma de la expresión a analizar:

- En el caso de un **literal**, la interpretación resultante son sencillamente siete conjuntos vacíos. Los literales son valores básicos y no apuntan a ninguna estructura y, por tanto, no comparten con ninguna otra variable del programa.

- La interpretación de una **variable** puede obtenerse a partir de las relaciones *SubR*, *ShR*, *Sub* y *Sh* acumuladas hasta el momento, independientemente del hecho de que se trate de una variable condenada que se está reutilizando (x!), o de una variable no condenada.

- Una expresión de la forma $x@r$ provoca la copia de la parte recursiva de la estructura apuntada por x en una región r distinta de la original. Por tanto, la estructura resultado de $x@r$ no es hija de ninguna otra estructura. Además, dado que los hijos recursivos también han sido recién creados en la copia, no son compartidos por ninguna otra variable. Por otro lado, para el cálculo del conjunto *ShRP* debemos sólo considerar variables cuyo tipo sea distinto de x, ya que en caso contrario estaríamos considerando hijos recursivos del mismo x que, como ya se ha comentado, son distintos de las copias recién creadas en la región de destino. La función $type(x)$ devuelve el tipo Hindley-Milner de la variable x.

- La definición de la interpretación S cuando la expresión es una **aplicación** de función de la forma $g\ \overline{a_i}^m@r$ requiere obtener la signatura de compartición $\rho(g)$ de la función llamada. En la misma encontramos las relaciones de compartición

$$
\begin{array}{lcl}
S\ [\![c]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \\
S\ [\![x]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\{z \mid x \in SubR(z)\}, \\
 & & \{z \mid x \in ShR(z)\}, \\
 & & \{z \mid x \in Sub(z)\}, \\
 & & SubR(x), ShR(x), Sub(x), Sh(x)) \\
S\ [\![x!]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & S\ [\![x]\!]\ SubR\ ShR\ Sub\ Sh\ \rho \\
S\ [\![x@r]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\emptyset, \\
 & & \{z \mid x \in ShR(z) \wedge type(z) \neq type(x)\}, \\
 & & \emptyset,\emptyset,\emptyset, \\
 & & Sub(x) - SubR(x), Sh(x)) \\
S\ [\![g\ \overline{a_i}^m@r]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\{z \mid \exists j \in SubRPg.a_j \in SubR(z)\}, \\
 & & \{z \mid \exists j \in Shg.a_j \in ShR(z)\}, \\
 & & \{z \mid \exists j \in SubPg.a_j \in Sub(z)\}, \\
 & & \bigcup_j\{SubR(a_j) \mid j \in SubRg\}, \\
 & & \bigcup_j\{Sh(a_j) \mid j \in ShRg\}, \\
 & & \bigcup_j\{Sub(a_j) \mid j \in Subg\}, \\
 & & \bigcup_j\{Sh(a_j) \mid j \in Shg\})
\end{array}
$$

where $(SubRPg, ShRPg, SubPg, SubRg, ShRg, Subg, Shg) = \rho(g)$

$$
\begin{array}{lcl}
S\ [\![C\ \overline{a_i}^m@r]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\emptyset, \\
 & & \{z \mid \exists a_j \in ShR(z)\}, \\
 & & \emptyset, \\
 & & \bigcup_j\{SubR(a_j) \mid j \in RecPos(C)\}, \\
 & & \bigcup_j\{ShR(a_j) \mid j \in RecPos(C)\}, \\
 & & \bigcup_j\{Sub(a_j) \mid j \in \{1..m\}\}, \\
 & & \bigcup_j\{Sh(a_j) \mid j \in \{1..m\}\}) \\
S\ [\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (S\ [\![e]\!]\ SubR_2\ ShR_2\ Sub_2\ Sh_2\ \rho)\backslash\{x_1\}
\end{array}
$$

where $(SubRP_1, ShRP_1, SubP_1, SubR_1, ShR_1, Sub_1, Sh_1) = S\ [\![e_1]\!]\ SubR\ ShR\ Sub\ Sh\ \rho$

$$
\begin{array}{l}
SubR_2 = (SubR \cup [x_1 \mapsto SubR_1] \cup \{[z \mapsto \{x_1\}] \mid z \in SubRP_1\})^* \\
ShR_2 = ShR \cup [x_1 \mapsto ShR_1] \cup \{[z \mapsto \{x_1\}] \mid z \in ShRP_1\} \cup SubR_2 \\
Sub_2 = (Sub \cup [x_1 \mapsto Sub_1] \cup \{[z \mapsto \{x_1\}] \mid z \in SubP_1\})^* \\
Sh_2 = Sh \cup \{\{x_1\} \cup Sh_1\} \uplus (Sub_2 \cup ShR_2)
\end{array}
$$

$$
S\ [\![\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}]\!]\ SubR\ ShR\ Sub\ Sh\ \rho = \bigcup_i((S\ [\![e_i]\!]\ SubR_i\ ShR_i\ Sub_i\ Sh_i\ \rho)\backslash\{\overline{x_{ij}}^{n_i}\})
$$

where

$$
\begin{array}{l}
SubR_i = (SubR \cup [x \mapsto \{x_{ij} \mid j \in RecPos(C_i)\}] \\
\qquad\qquad \cup \{[x_{ij} \mapsto SubR(x)\backslash\{x\}] \mid j \in RecPos(C_i)\})^* \\
ShR_i = ShR \cup \{[x_{ij} \mapsto ShR(x)] \mid j \in RecPos(C_i)\} \cup SubR_i \\
Sub_i = (Sub \cup [x \mapsto \{x_{ij} \mid j \in \{1..n_i\}] \cup \{[x_{ij} \mapsto Sub(x)\backslash\{x\}] \mid j \in \{1..n_i\}\})^* \\
Sh_i = (Sh \cup \{\{y, x_{ij}\} \mid y \in Sh(x) \wedge j \in \{1..n_i\}\}) \uplus (Sub_i \cup ShR_i)
\end{array}
$$

Figura 4.2: Definición de la interpretación abstracta S

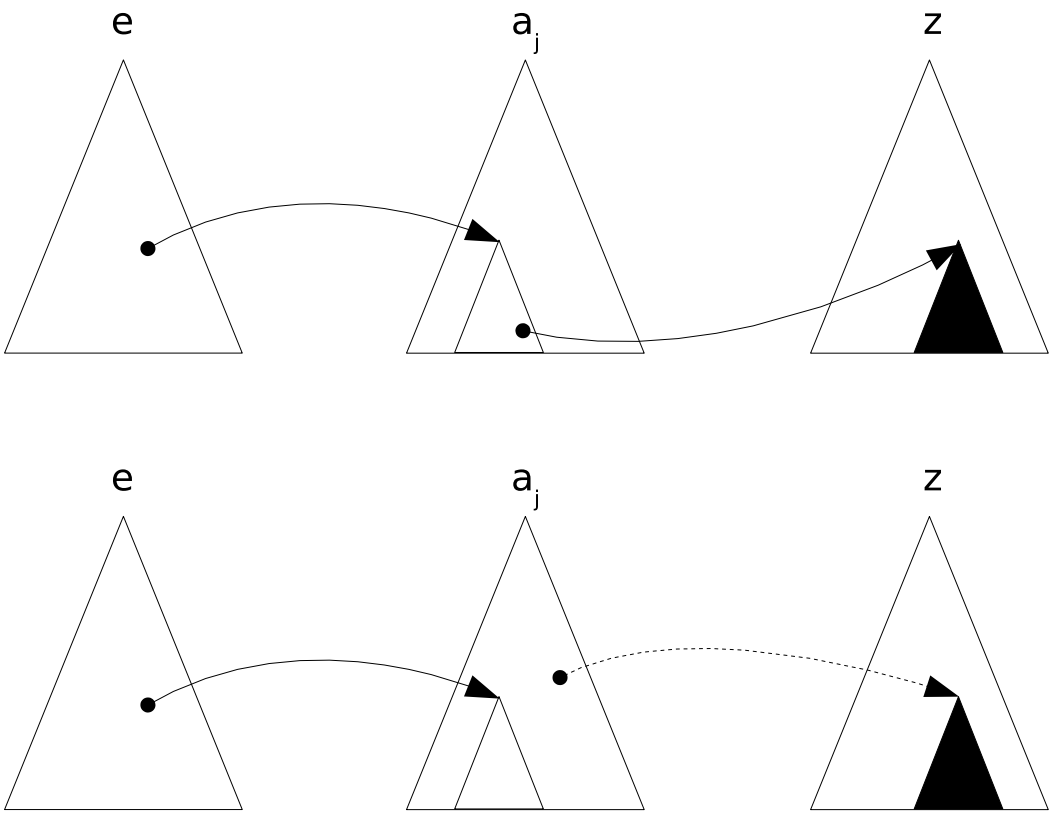

Figura 4.3: Relación de compartición *ShR* en una aplicación de función. Sea e la variable anónima que apunta al resultado que devuelve la función llamada. En la figura de arriba tenemos que realmente $e \mathrel{\triangle\!\!\sim} z$. Sin embargo, en la figura de abajo esto no es así.

del resultado de la función con respecto a los parámetros. De este modo, podemos aplicar la transitividad de las relaciones $\lhd$ y $\lhd\!\!\sim$ para obtener el primer y tercer conjunto de la interpretación. Si el resultado es hijo (recursivo) de alguno de los parámetros, entonces también será hijo (recursivo) de aquellas variables de las que dichos parámetros sean hijos (recursivos). Un razonamiento dual permite obtener las expresiones para los conjuntos cuarto y sexto. Con respecto al segundo conjunto, el resultado compartirá un hijo recursivo de una variable z si existe un parámetro real a_j que comparta un hijo recursivo de z y el resultado de la aplicación de la función (conforme indica la signatura para la misma) comparte cualquier estructura con el parámetro. En este caso tenemos una pérdida de precisión en el análisis, provocando que en este conjunto aparezcan más variables de las que realmente tengan un hijo recursivo al que el resultado de la llamada apunte (Figura 4.3). No obstante, se trata de una aproximación segura. Esto se aplica del mismo modo a los conjuntos quinto y séptimo del resultado.

- En el caso de la **construcción** de una estructura de datos, la estructura creada no forma parte en principio de ninguna estructura, por lo que los conjuntos primero y tercero quedan vacíos. Para el resto de componentes del resultado basta con obtener las relaciones correspondientes de cada una de las variables con las que se está construyendo la estructura y reunir los resultados. Cabe destacar que para las relaciones *SubR* y *ShR* (conjuntos cuarto y quinto, respectivamente) sólo debemos considerar las variables que aparezcan en posiciones recursivas de la constructora; el conjunto $RecPos(C)$ contiene dichas posiciones.

- Para la construcción **let** consideramos primero la interpretación de la expresión auxiliar e_1. Una vez hecho esto, utilizamos la información obtenida para actualizar las relaciones acumuladas para incluir la variable x_1. Con esto obtenemos las relaciones $SubR_2$, ShR_2, Sub_2 y Sh_2. El operador $\uplus$ calcula la unión simétrica de dos relaciones, mientras que con la notación R^* hacemos referencia al cierre reflexivo y transitivo de la relación R. Nótese que en la definición de ShR_2 se realiza la unión de la relación $SubR_2$ y en la definición de Sh_2 se añaden las relaciones Sub_2 y ShR_2. Esto es consecuencia de la implicación entre las distintas relaciones, vista anteriormente.

- Por último, la interpretación de una expresión **case** es el supremo de las interpretaciones de cada alternativa por separado, lo cual conlleva una pérdida de precisión. Antes de proceder a analizar cada alternativa, debemos completar las relaciones acumuladas con la información de las variables involucradas en la guarda de la misma. Con ello se obtienen las relaciones $SubR_i$, ShR_i, Sub_i y Sh_i. La interpretación del case destructivo (**case**!) se obtiene de la misma forma.

4.3. Análisis de las definiciones

En la interpretación abstracta S descrita anteriormente se mantienen las signaturas de las funciones en un entorno ρ. En esta sección se describe el análisis de las definiciones de funciones que conforman el programa y la actualización de dicho entorno de signaturas a medida que se evalúan las relaciones de compartición de las declaraciones.

La interpretación S aplicada a una definición $f\ x_1 \ldots x_n = e$ obtiene como resultado una signatura. Como se dijo anteriormente, esta signatura indica las relaciones de compartición del resultado de f con respecto a los parámetros de entrada x_i, y tiene la siguiente forma:

$$(S_1, S_2, S_3, S_4, S_5, S_6, S_7) \qquad \text{donde } \forall i \in \{1 \ldots 7\}.\ S_i \subseteq \{1 \ldots n\}$$

El significado de cada conjunto es el mismo que en el caso de la interpretación de las expresiones vista en la Sección 4.2, con la diferencia de que los conjuntos de la signatura sólo contienen los *índices* de los parámetros, en lugar de las variables en ámbito.

Si queremos obtener la signatura de una definición $f\ x_1 \ldots x_n = e$ es necesario en primer lugar hallar la interpretación de la expresión e. La obtención de la signatura a partir de este resultado resulta sencilla; definimos la función *extract* de este modo:

$$\begin{aligned} &\mathit{extract}(xs, (S_1, \ldots, S_7)) = (\{i \mid x_i \in xs \cap S_1\}, \ldots, \{i \mid x_i \in xs \cap S_7\}) \\ &\quad \textbf{donde } xs = [x_1, \ldots, x_n] \end{aligned}$$

Como se explicó anteriormente, para calcular la interpretación S de una expresión es necesario mantener y acumular las relaciones de compartición entre las variables que van apareciendo en ámbito. En el momento de comenzar el análisis de cada expresión la única información disponible que tenemos es el hecho de que cada parámetro de entrada está relacionado consigo mismo en cada una de las cuatro relaciones $\lhd\!\sim$, $\triangle\!\sim$, $\lhd$ y $\triangle$, ya que todas son reflexivas. De este modo definimos las relaciones iniciales $SubR_0$, ShR_0, Sub_0 y Sh_0:

$$\begin{aligned} &SubR_0 = ShR_0 = Sub_0 = \{[x_i \mapsto \{x_i\}] \mid i \in \{1 \ldots n\}\} \\ &Sh_0 = \{\{x_i\} \mid i \in \{1 \ldots n\}\} \end{aligned}$$

La interpretación de una definición recibe un entorno de signaturas y añade la signatura de la nueva definición al entorno:

$$\begin{aligned} &S\ [\![f\ \overline{x_i}^n = e]\!]\ \rho\ =\ \mathit{FIX}\ F\ \rho_0 \\ &\quad \textbf{donde}\ \ F = \lambda\rho.\rho[f \mapsto \mathit{extract}([x_1, \ldots, x_n], S\ [\![e]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho)] \\ &\qquad\qquad \rho_0 = \rho[f \mapsto (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)] \end{aligned}$$

donde $\rho[f \mapsto s]$ añade la signatura s para la función f, o la reemplaza en el caso en que ya existiese. Es necesario el cálculo del menor punto fijo, ya que la función f puede ser recursiva y, por tanto, se necesitaría en el propio análisis la propia signatura de f. Dado que el conjunto de signaturas junto con el orden de inclusión entre cada componente de la tupla conforman un retículo finito y las funciones S y *extract* son monótonas, el punto fijo existe y puede calcularse mediante la cadena ascendente de Kleene:

$$\mathit{FIX}\ F\ \rho_0 = \bigsqcup_{n \in \mathbb{N}} F^n(\rho_0)$$

4.4. Análisis de un programa

Dado el siguiente programa:

$$P = dec_1; dec_2; \ldots; dec_n; e$$

donde dec_i son las declaraciones de funciones y e es la expresión principal que calcula el resultado del programa, el análisis de compartición se realiza del siguiente modo: A medida que se realiza el análisis de las definiciones obtenemos un entorno de signaturas que se va ampliando con cada definición. Posteriormente se analizará la expresión principal.

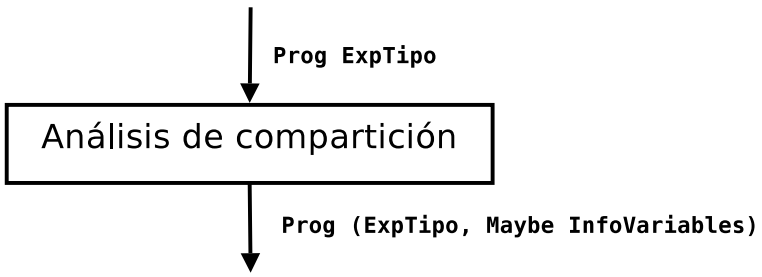

Figura 4.4: Entrada y salida del análisis de compartición

La interpretación S queda definida del siguiente modo para los programas:

$$S\ [\![P]\!] = S\ [\![e]\!]\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ (S\ [\![dec_n]\!]\ (\ldots(S\ [\![dec_1]\!]\ [\,])\ldots))$$

4.5. Detalles de implementación

En esta sección se explicarán algunos aspectos de la implementación del análisis de compartición y su integración en el resto de fases del compilador. La implementación está en correspondencia directa con la definición de la función S mostrada en las secciones anteriores.

El análisis consiste en un recorrido del árbol abstracto del programa proveniente de la fase de transformación de *Full-Safe* a *Core-Safe*. Cada definición, patrón y expresión viene decorada con su tipo Hindley-Milner correspondiente. Esto es necesario para el cálculo de la interpretación S para expresiones de la forma $x@r$ (ver Figura 4.2), donde se requiere saber el tipo de las variables que en ese momento están en ámbito.

Por otro lado, el análisis devuelve el árbol abstracto del programa decorado con pares `(ExpTipo, Maybe InfoVariables)`. La primera componente del par contendrá el mismo tipo Hindley-Milner proveniente de la fase anterior, sin modificar. La segunda componente contendrá información de compartición de variables, tal como queda almacenada en la estructura de datos *Relaciones* (Ver Sección 4.5.1). Esta información, que resultará necesaria en la fase de inferencia de tipos SAFE (Capítulo 5), se detallará más en la Sección 4.5.5.

4.5.1. Estructura de datos *Relaciones*

Durante el recorrido del programa se realizan de forma frecuente accesos a las cuatro relaciones definidas anteriormente (esto es, $\lhd\!\sim$, $\triangle\!\sim$, $\lhd$ y $\triangle$). Para hacer esto eficientemente se ha implementado una estructura de datos `Relaciones` encargada de almacenar dichas relaciones. Esta estructura hace uso de los módulos `Data.Map` y `Data.Set` de las librerías de GHC [Ada93].

```
type Relaciones = M.Map Variable InfoVariable
```

Nombre	Descripción
`SubRDir`	Conjunto de variables z tales que $z \lhd\!\sim x$
`SubRInv`	Conjunto de variables z tales que $x \lhd\!\sim z$
`ShRDir`	Conjunto de variables z tales que $z \triangle\!\sim x$
`ShRInv`	Conjunto de variables z tales que $x \triangle\!\sim z$
`SubDir`	Conjunto de variables z tales que $z \lhd x$
`SubInv`	Conjunto de variables z tales que $x \lhd z$
`ShDir`	Conjunto de variables z tales que $x \triangle z$

Cuadro 4.1: Información almacenada por cada variable x del programa

En el primer nivel disponemos de una tabla, implementada como un único árbol equilibrado, que asocia cada nombre de variable con la información de compartición relativa a ésta. Dicha información se encuentra almacenada como un tipo de datos `InfoVariable`. Para acceder a la información correspondiente a una determinada variable podemos utilizar la función `infoVariable`:

```
infoVariable :: Relaciones -> Variable -> InfoVariable
```

Ahora bien, nos queda saber cual es la estructura interna del tipo `InfoVariable`. Cada objeto de este tipo se compone de una tupla de siete componentes (Cuadro 4.1). Todas ellas son conjuntos implementados también como árboles equilibrados. El acceso a una de estas componentes es realizado mediante la función `obtenerRelacion`:

```
obtenerRelacion :: Selector -> InfoVariable -> Variables
```

donde `Selector` queda definido de este modo:

```
data Selector = SubRDir | SubRInv | ShRDir | ShRInv | SubDir
              | SubInv | ShDir
```

Para crear una estructura `Relaciones` disponemos de la función `consRelaciones` que, a partir de una lista con todas las variables libres que aparecen en una definición, devuelve una estructura `Relaciones` inicializada en la que todas las variables aparecen relacionadas consigo mismas.

Cabe señalar que por cada variable no sólo se almacenan las relaciones $\lhd\!\sim$, $\triangle\!\sim$, $\lhd$ y $\triangle$ sino también sus inversas, salvo en el caso de $\triangle$, que es simétrica. Esta redundancia se produce por motivos de eficiencia: a lo largo del análisis resulta con frecuencia necesario conocer, por ejemplo, no sólo los hijos recursivos de x sino además aquellas variables de las que x es hijo recursivo. Asociando las relaciones inversas a cada variable podemos obtenerlos de forma directa. No obstante, además del evidente aumento del coste en espacio esto conlleva un coste adicional de mantenimiento de la estructura: cada vez que se modifica una relación tenemos también que modificar la inversa. La actualización de una relación se lleva a cabo mediante la función `anadirRelacion`:

SubR	$x \leftsquigarrow y$
ShR	$x \triangle\!\!\!\leftsquigarrow y$
Sub	$x \lhd y$
Sh	$x \triangle y$

Cuadro 4.2: Valores permitidos en la función `anadirRelacion`

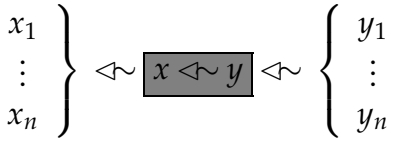

Figura 4.5: Caso peor en la adición de una relación $x \leftsquigarrow y$

```
anadirRelacion :: Relacion -> Variable -> Variable -> Relaciones
               -> Relaciones
```

en la que se indican las variables x e y que quieren relacionarse y el tipo de relación que quiere añadirse (Cuadro 4.2). En la implementación de la función se debe mantener la posible simetría y transitividad de la relación modificada. Para ello:

- Si se añade la información $x \triangle y$, debemos añadir también $y \triangle x$
- Si se añade la información $x \triangle\!\!\!\leftsquigarrow y$, debemos añadir también $y(\triangle\!\!\!\leftsquigarrow)^{-1}x$
- Si se añade la información $x \lhd y$, consideramos los conjuntos:

  $$X = \{z | z \lhd x\} \qquad Y = \{z | y \lhd z\}$$

  Para calcular el cierre transitivo de $\lhd$, tendremos que añadir $x' \lhd y$ e $y(\lhd)^{-1}x'$ para todo $x' \in X$ y añadir $x \lhd y'$ e $y'(\lhd)^{-1}x$ para todo $y' \in Y$.
- Si se añade la información $x \leftsquigarrow y$, debemos tener en cuenta las mismas consideraciones que en el caso anterior.

Consideremos el caso peor para relacionar dos variables x e y mediante $\leftsquigarrow$ (Figura 4.5). En este caso todas las las variables contenidas en la estructura son hijas recursivas de x y todas las variables de la estructura tienen a y como hijo recursivo. Se ha de añadir el conjunto $\{x_1, \ldots, x_n\}$ a en la información $(\leftsquigarrow)$ de cada variable del conjunto $\{y_1, \ldots, y_n\}$. Además es necesario añadir el conjunto $\{y_1, \ldots, y_n\}$ a la información $(\leftsquigarrow)^{-1}$ de cada elemento del conjunto $\{x_1, \ldots, x_n\}$. Tenemos $2n$ uniones entre conjuntos de n elementos. Teniendo en cuenta que la unión de dos conjuntos de cardinales n y m respectivamente tiene coste $\mathcal{O}(n+m)$, la adición de esta nueva relación entre x e y tiene coste $\mathcal{O}(n^2)$, siendo n el número de variables en la estructura.

Además de añadir una relación entre dos variables cualesquiera del programa, en el análisis también resulta a veces necesaria la unión de dos relaciones. Para ello se dispone de la función `unirRelaciones`

```
unirRelaciones :: [Selector] -> Selector -> Relaciones -> Relaciones
```

donde, para cada variable de la estructura, se reúnen en un solo conjunto todas las variables correspondientes a la lista de selectores y se almacena el resultado en otra componente. Se debe tener en cuenta que esta función usada de modo incorrecto (por ejemplo, uniendo $\lhd$ y $\triangle\!\!\!\sim$ pero no sus inversas) puede romper las propiedades de transitividad y simetría de algunas relaciones, por lo que el usuario de la función es el responsable de escoger los selectores adecuados para no perder dichas propiedades.

4.5.2. Información de programa

A medida que se realiza el recorrido descendente del árbol abstracto para calcular la información de compartición de las expresiones resulta necesaria cierta información que en la definición de la interpretación S se considera "global". Un ejemplo de esto es la función *RecPos* que aparece en la definición de S para expresiones **case** (ver Figura 4.2).

Para propagar esta información resulta necesaria la inclusión de un parámetro adicional en la función `comparticionExp`, encargada del análisis de expresiones. Este parámetro contiene una estructura de tipo `InfoPrograma`, definida del siguiente modo:

```
type InfoPrograma = (TablaRecPos, TablaTipoVariables, Entorno)
```

La primera de las componentes de esta tupla es en una tabla que asocia a cada constructor una lista de números naturales, que indica las posiciones recursivas del tipo que se está definiendo. Esta tabla se construye antes del análisis de compartición, examinando las declaraciones `data` del programa. Por ejemplo, dadas las siguientes definiciones de estructuras de datos[1]:

```
data Lista a = Nil | Cons a (Lista a)
data Arbol a = Vacio | Nodo (Arbol a) a (Arbol a)
```

Se obtiene la siguiente tabla de posiciones recursivas:

$$\begin{bmatrix} \texttt{Nil} & \mapsto & [\,] \\ \texttt{Cons} & \mapsto & [2] \\ \texttt{Vacio} & \mapsto & [\,] \\ \texttt{Nodo} & \mapsto & [1,3] \end{bmatrix}$$

Para la implementación de la estructura `TablaRecPos` se hace uso del módulo `Data.Map` de las librerías de GHC:

```
type TablaRecPos = M.Map String [Int]
```

[1] Se omiten variables de región, por motivos de claridad

La segunda componente de la tupla `InfoPrograma` contiene una tabla que indica el tipo de cada variable de una declaración. Esta tabla se construye realizando un recorrido del árbol abstracto antes de analizar cada declaración. Cada variable encontrada se añade a esta tabla junto con su tipo (que se encuentra en la decoración del árbol abstracto). Al igual que la estructura anterior, esta tabla también se cuentra implementada con los módulos disponibles en las librerías de GHC:

```
type TablaTipoVariables = M.Map Variable ExpTipo
```

En último lugar, se incluye también como parte de la estructura `InfoPrograma` el entorno ρ que se propaga como parámetro en la interpretación S. Esta estructura, de tipo `Entorno`, es una tabla que asocia cada nombre de función con su signatura de compartición, y se encuentra definida del siguiente modo:

```
type Entorno = M.Map NombreFuncion SignaturaFuncion
```

Donde una `SignaturaFuncion` es una tupla formada por siete listas ordenadas de enteros, donde se indica las relaciones de compartición del resultado con respecto a sus parámetros:

```
type SignaturaFuncion = ([Int], [Int], [Int], [Int], [Int], [Int], [Int])
```

Junto con el tipo `InfoPrograma` se incluyen algunas operaciones de acceso a los elementos de estas tablas, cuyas signaturas se indican a continuación:

```
infoRecPos    :: InfoPrograma -> String -> [Int]
infoTipoVar   :: InfoPrograma -> Variable -> ExpTipo
infoSignatura :: InfoPrograma -> NombreFuncion -> SignaturaFuncion
```

También se definen funciones modificadoras de la información de programa, concretamente para establecer la tabla de tipos de variables previamente construida y para añadir una nueva entrada al entorno de compartición:

```
infoGuardarSignatura :: NombreFuncion -> SignaturaFuncion
                     -> InfoPrograma -> InfoPrograma

infoGuardarTablaTipoVariables :: TablaTipoVariables
                              -> InfoPrograma -> InfoPrograma
```

4.5.3. Análisis de expresiones

La interpretación S definida para las expresiones tal como se muestra en la Figura 4.2 queda implementada por la función `comparticionExp`, la finalidad de esta función es doble:

- Por un lado, calcular las relaciones de compartición de acuerdo con la definición de S. El resultado se devolverá en una estructura de tipo `InfoCompparticion`.

- Por otra parte, actualizar la decoración del árbol abstracto, incluyendo la información de compartición de cada variable, tal como se explica en la Sección 4.5.5.

La signatura de esta función se muestra a continuación:

```
comparticionExp :: Relaciones -> InfoPrograma
                -> Exp ExpTipo
                -> (Exp (ExpTipo, Maybe InfoVariables), InfoComparticion)
```

El primer parámetro, de tipo `Relaciones` contiene la estructura de datos que almacena las relaciones entre variables, de la que se explicó su funcionamiento en la Sección 4.5.1. La información de esta estructura se va ampliando durante el recorrido descendente del árbol abstracto, a medida que aparecen nuevas variables en ámbito. El segundo parámetro contiene las tablas de posiciones recursivas, tipos de variables y signaturas de función necesarias para el cálculo de S.

La función `comparticionExp` devuelve como resultado el árbol abstracto de la expresión con la decoración actualizada y el resultado del análisis, que se corresponde con los siete conjuntos del resultado de la interpretación S.

Tanto los tipos de datos `InfoVariable` como `InfoComparticion` están definidos como una tupla de siete conjuntos. Los datos `InfoVariable` forman parte de la estructura *Relaciones* y acumulan la información relativa a cada variable que aparece libre en la expresión. Los datos `InfoComparticion` contienen el resultado de la interpretación S. Mientras que los datos `InfoVariable` se asocian a variables, los datos `InfoComparticion` son el resultado del análisis de las expresiones del programa.

```
type Variables = S.Set Variable
type InfoComparticion = (Variables, Variables, Variables, Variables,
                         Variables, Variables, Variables)
```

La implementación de la función `comparticionExp` es una traducción directa de las ecuaciones de la Figura 4.2. Se incluye, a modo de ejemplo, el código correspondiente al análisis de expresiones de la forma **let** $x_1 = e_1$ **in** e y la ecuación correspondiente para S. Ambos se encuentran en la Figura 4.6. En primer lugar se analiza y decora la expresion auxiliar e_1, obteniendo como resultado los conjuntos `(subRP1, shRP1, ..., sh1)`. A continuación se incorporan sucesivamente las relaciones de la variable x_1 en la estructura *Relaciones*. La variable `rels1` comprende la ecuación de $SubR_2$, las variables `rels2` y `rels3` incorporan la información que corresponde con la ecuación ShR_2. Por su parte, la variable `rels4` comprende la expresión asociada a Sub_2 mientras que `rels5` y `rels6` se corresponden con la ecuación de Sh_2. Con la estructura de relaciones actualizada podemos analizar la expresión principal e, obteniendo como resultado `infoComp'`, que es la interpretación S de la expresión **let** original.

4.5.4. Análisis del programa

La interpretación abstracta S para una declaración de función requiere la construcción previa de la tabla de tipos de variables que aparecen en la misma. Una vez hecho

```
comparticionExp' rels ip
          (LetE [(ets, (x1, patBools,vregs), Simple e1 [])] e tipo)
   = (LetE [(ets, (x1, patBools',vregs), Simple e1' [])] e'
             (tipo, Just infoVariables, infoComp')
    where -- Análisis de e1
          (e1',(subRP1, shRP1, subP1, subR1, shR1, sub1, sh1)) =
                  comparticionExp rels ip e1
          -- Construcción de relación
          rels1 = anadirRelacionesConjVar SubR subR1 x1
                        (anadirRelacionesVarConj SubR x1 subRP1 rels)
          rels2 = anadirRelacionesConjVar ShR shR1 x1
                        (anadirRelacionesVarConj ShR x1 shRP1 rels1)
          rels3 = unirRelaciones [SubRInv, ShRInv] ShRInv
                        (unirRelaciones [SubRDir, ShRDir] ShRDir rels2)
          rels4 = anadirRelacionesConjVar Sub sub1 x1
                        (anadirRelacionesVarConj Sub x1 subP1 rels3)
          rels5 = anadirRelacionesVarConj Sh x1 sh1 rels4
          rels6 = unirRelaciones [SubDir, SubInv, ShRDir, ShRInv, ShDir]
                        ShDir rels5
          -- Análisis de e
          (e', infoComp) = comparticionExp rels6 ip e
          -- Decoración de los patrones y booleanos
          patBools' = [(comparticionPatron p,b) | (p,b) <- patBools]
          -- Quitamos la variable ligada del resultado antes de devolverlo
          infoComp' = mapInfoComparticion (S.delete x1) infoComp
          -- Decoración que será incluida con la expresión
          infoVariables = ...
```

$$S\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e]\!]\ SubR\ ShR\ Sub\ Sh\ \rho \quad = \quad (S\ [\![e]\!]\ SubR_2\ ShR_2\ Sub_2\ Sh_2\ \rho) \backslash \{x_1\}$$

$$\begin{aligned}
&\mathbf{where}\ (SubRP_1, ShRP_1, SubP_1, SubR_1, ShR_1, Sub_1, Sh_1) = S\ [\![e_1]\!]\ SubR\ ShR\ Sub\ Sh\ \rho \\
&\quad SubR_2 = (SubR \cup [x_1 \mapsto SubR_1] \cup \{[z \mapsto \{x_1\}] \mid z \in SubRP_1\})^* \\
&\quad ShR_2 = ShR \cup [x_1 \mapsto ShR_1] \cup \{[z \mapsto \{x_1\}] \mid z \in ShRP_1\} \cup SubR_2 \\
&\quad Sub_2 = (Sub \cup [x_1 \mapsto Sub_1] \cup \{[z \mapsto \{x_1\}] \mid z \in SubP_1\})^* \\
&\quad Sh_2 = Sh \cup \{\{x_1\} \cup Sh_1\} \uplus (Sub_2 \cup ShR_2)
\end{aligned}$$

Figura 4.6: Código y ecuación de S para expresiones **let**

esto, se procede a hacer el análisis de la expresión principal de la definición, obteniendo siete conjuntos de variables. La signatura de compartición de la función puede calcularse a partir de estos conjuntos y la lista de variables que conforman los parámetros de la definición. La función `extract` es la encargada de convertir el conjunto de variables en una lista ordenada de posiciones.

```
extract :: [Patron a] -> Variables -> [Int]
extract pats vars = [ind | (ind, VarP var _) <- zip [1..] pats,
                                                S.member var vars]
```

El análisis de la expresión principal de una definición se realiza repetidamente mediante la función `comparticionDef'`, que va actualizando la signatura y la decoración del árbol abstracto hasta que se alcanza un punto fijo. Por su parte, la función `comparticionDef` realiza la llamada inicial a la anterior, construyendo previamente la tabla de tipos de variables.

```
comparticionDef :: InfoPrograma -> Def ExpTipo ->
                   (InfoPrograma, Def (ExpTipo, Maybe InfoCompartición))
```

La función `comparticionProg` realiza el análisis de todo el programa y se corresponde con la definición de S comentada en la Sección 4.4. En primer lugar se construye la tabla con las posiciones recursivas de cada constructora para poder formar la estructura `InfoPrograma` inicial. Una vez hecho esto, se proceden a analizar todas las definiciones, ampliando progresivamente el entorno, para después realizar el análisis de la expresión principal del programa.

```
comparticionProg' :: Prog ExpTipo
                   -> Prog (ExpTipo, Maybe InfoCompartición)
comparticionProg' (decsData, defs, exp) = (decsData, defs', exp')
    where (_, defs') = L.mapAccumL comparticionDef ipInicial defs
          ipInicial = (construirTablaRecPos decsData, M.empty, ent)
          (exp',_) =
                comparticionExp (consRelaciones (variablesExp exp))
                ipFinal exp
```

4.5.5. Decoración del árbol abstracto

Ya ha sido explicado que, tras el análisis de compartición, cada nodo del árbol abstracto queda decorado con un par `(ExpTipo, Maybe InfoVariables)`. La segunda componente de este par contendrá en algunos nodos una colección de datos de tipo `InfoVariable`, cada uno de ellos asociado a una variable. La información que se asocia es la acumulada hasta el momento en la estructura de datos *Relaciones*. Cada información concreta (hijos recursivos, variables que comparten un hijo recursivo, etc.) podrá luego ser obtenida para cada variable mediante la función `obtenerRelacion`, descrita en la Sección 4.5.1.

La información realmente almacenada en esta segunda componente de la decoración depende del tipo de nodo. Por ejemplo, en las expresiones de la forma **let** $x_1 =$

e_1 **in** e la decoración incluirá la información de la variable x_1 acumulada en la estructura *Relaciones* justo en el momento antes de analizar e, tras haber analizado la expresión e_1. Por otra parte, en el caso de expresiones de la forma $x!$, $x@r$, $f\ \overline{a_i}^n@r$, $C\ \overline{a_i}^n@r$ y **case** no destructivo, la decoración será `Nothing`.

El motivo por el cual se incluye la información de compartición sólo en ciertos tipos de expresiones y no se incluye en el resto se verá más adelante en el algoritmo de inferencia de tipos SAFE (Capítulo 5). En determinadas reglas de esta inferencia encontraremos referencias a dos funciones llamadas *sharerec* y *shareall*:

- $sharerec(x, e)$: devuelve el conjunto de variables en ámbito en e que en tiempo de ejecución podrían apuntar a un hijo recursivo de la estructura a la que apunta x.
- $shareall(x, e)$: devuelve el conjunto de variables en ámbito en e que en tiempo de ejecución podrían apuntar a un hijo de la estructura a la que apunta x.

La información devuelta por las funciones *sharerec* y *shareall* puede obtenerse a partir de la decoración `InfoVariables` incorporada en el árbol abstracto en esta etapa de análisis. En aquellas partes del árbol abstracto donde no se utilizaría esta decoración (ya que en las reglas de inferencia correspondiente no habría ninguna referencia a *shareall* ni *sharerec*) incluimos el valor `Nothing` como decoración.

4.6. Ejemplos

A continuación se ilustrará paso a paso mediante varios ejemplos el análisis explicado a lo largo de este capítulo. Comenzaremos considerando una definición de la función *fst*, que devuelve la primera componente de un par. Aunque en *Core-Safe* los pares son representadas mediante la constructora *Tupla* (por ejemplo *Tupla* $x_1\ x_2$), en el código se utilizará la notación (x_1, x_2) por motivos de claridad. El código *Core-Safe* correspondiente a esta función se muestra a continuación:

$$\begin{array}{|l} fst\ p = \textbf{case}\ p\ \textbf{of} \\ \qquad\qquad (x_1, x_2) \rightarrow x_1 \end{array}$$

El objetivo es el cálculo de $S\ [\![\textbf{case}\ p\ \textbf{of} \ldots]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho_0$, donde:

$$\begin{array}{l} SubR_0 = ShR_0 = Sub_0 = [p \mapsto \{p\}] \\ Sh_0 = \{\{p\}\} \\ \rho_0 = [fst \mapsto (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)] \end{array}$$

Es decir, partimos de una estructura *Relaciones* con una única variable libre p relacionada consigo misma (ya que las cuatro relaciones son reflexivas) y un entorno inicial en el que la signatura asociada a *fst* son siete conjuntos vacíos. Antes de analizar el cuerpo de la única alternativa de la expresión **case** se han de actualizar las relaciones incorporando la información de variables ligadas en la guarda de dicha alternativa. Esto dará lugar a las siguientes realciones de compartición:

$$\begin{array}{rcl} SubR_1 & = & [p \mapsto \{p\}, x_1 \mapsto \{x_1\}, x_2 \mapsto \{x_2\}] \\ ShR_1 & = & [p \mapsto \{p\}, x_1 \mapsto \{x_1\}, x_2 \mapsto \{x_2\}] \\ Sub_1 & = & [p \mapsto \{p, x_1, x_2\}, x_1 \mapsto \{x_1\}, x_2 \mapsto \{x_2\}] \\ Sh_1 & = & \{\{p\}, \{p, x_1\}, \{p, x_2\}\} \end{array}$$

Para realizar el cálculo de la interpretación para el cuerpo de la alternativa, esto es, $S\ [\![x_1]\!]\ SubR_1\ ShR_1\ Sub_1\ Sh_1\ \rho_0$, basta con acceder a las relaciones que se han acumulado hasta el momento. El resultado es $(\{x_1\}, \{x_1\}, \{x_1, p\}, \{x_1\}, \{x_1\}, \{x_1\}, \{x_1, p\})$ que corresponde a la signatura $(\emptyset, \emptyset, \{1\}, \emptyset, \emptyset, \emptyset, \{1\})$. Repitiendo el análisis con el entorno ρ_1, en el que la función *fst* queda asociada con esta nueva signatura, obtenemos el mismo resultado. Hemos alcanzado el punto fijo, con lo que el entorno final es:

$$\boxed{fst \mapsto (\emptyset, \emptyset, \{1\}, \emptyset, \emptyset, \emptyset, \{1\})}$$

Es decir, el resultado devuelto por la función es hijo no recursivo del primer parámetro. El valor 1 del último conjunto de la signatura es también consecuencia directa de ello.

A continuación consideraremos la definición de las funciones *revD* y *revauxD*, que implementan la inversión del orden de los elementos de una lista mediante un parámetro acumulador. La sintaxis *Core-Safe* correspondiente se muestra a continuación:

```
revD xs @r = let x1 = [ ]@r in (revauxD xs x1) @r

revauxD xs ys @r =
        case! xs of
              [ ] → ys
              (x : xx) → let x1 = (x : ys)@r in (revauxD xx x1)@r
```

Empezamos con la definición de *revauxD*. A partir de los parámetros de entrada construimos la estructura *Relaciones* inicial, que representaremos mediante la tupla $(SubR_0, ShR_0, Sub_0, Sh_0)$ donde:

$$\begin{array}{rcl} SubR_0 & = & [xs \mapsto \{xs\}, ys \mapsto \{ys\}] \\ ShR_0 & = & [xs \mapsto \{xs\}, ys \mapsto \{ys\}] \\ Sub_0 & = & [xs \mapsto \{xs\}, ys \mapsto \{ys\}] \\ Sh_0 & = & \{\{xs\}, \{ys\}\} \end{array}$$

De este modo, podemos iniciar el cálculo del punto fijo a partir del entorno inicial $\rho_0 = [revauxD \mapsto (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)]$. Calculamos $S[\![e]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho_0$, siendo e el lado derecho de la definición de *revauxD*.

La expresión e es de la forma **case**! xs **of** ... con dos alternativas. Por tanto, se examinará cada alternativa por separado para después realizar la unión de los dos resultados.

- **Alternativa** $[\,]$

  Dado que no hay variables ligadas en esta alternativa no es necesario añadir ninguna información a las relaciones acumuladas hasta el momento. Por tanto, calculamos $S\ [\![ys]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho_0$. El resultado obtenido es:

$$(\{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\})$$

- **Alternativa** $(x : xx)$

  Antes de analizar la expresión correspondiente a esta alternativa, incluimos la información de x y xx en las relaciones de compartición, obteniendo:

$$\begin{array}{rcl} SubR_1 & = & [xs \mapsto \{xs, xx\}, ys \mapsto \{ys\}, xx \mapsto \{xx\}, x \mapsto \{x\}] \\ ShR_1 & = & [xs \mapsto \{xs, xx\}, ys \mapsto \{ys\}, xx \mapsto \{xx\}, x \mapsto \{x\}] \\ Sub_1 & = & [xs \mapsto \{xs, xx, x\}, ys \mapsto \{ys\}, xx \mapsto \{xx\}, x \mapsto \{x\}] \\ Sh_1 & = & \{\{xs\}, \{ys\}, \{xs, xx\}, \{xs, x\}\} \end{array}$$

  Ahora pasamos a calcular $S\ [\![\mathbf{let}\ x_1 = \ldots\ \mathbf{in}\ \ldots]\!]\ SubR_1\ ShR_1\ Sub_1\ Sh_1\ \rho_0$; el análisis de la expresión auxiliar del **let** da como resultado los siguientes conjuntos:

$$(\emptyset, \{x, ys\}, \emptyset, \{ys\}, \{ys\}, \{x, ys\}, \{x, xs, ys\})$$

  A partir de los cuales podemos acumular la información de x_1 en las relaciones de compartición para obtener las relaciones $(SubR_2, ShR_2, Sub_2, Sh_2)$ necesarias en la interpretación de la expresión principal del **let**.

$$\begin{array}{rcl} SubR_2 & = & [xs \mapsto \{xs, xx\}, ys \mapsto \{ys\}, xx \mapsto \{xx\}, x \mapsto \{x\}, x_1 \mapsto \{x_1, ys\}] \\ ShR_2 & = & [xs \mapsto \{xs, xx\}, ys \mapsto \{ys, x_1\}, xx \mapsto \{xx\}, x \mapsto \{x, x_1\}, x_1 \mapsto \{x_1, ys\}] \\ Sub_2 & = & [xs \mapsto \{xs, xx, x\}, ys \mapsto \{ys\}, xx \mapsto \{xx\}, x \mapsto \{x\}, x_1 \mapsto \{x_1, x, ys\}] \\ Sh_2 & = & \{\{xs\}, \{ys\}, \{xs, xx\}, \{xs, x\}, \{x_1, xs, x, ys\}\} \end{array}$$

  Para el cálculo de $S\ [\![(revauxD\ xx\ x_1)@r]\!]\ SubR_2\ ShR_2\ Sub_2\ Sh_2\ \rho_0$ y considerando la signatura inicial para *revauxD*, tenemos que el resultado de la interpretación es:

$$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

  que es el resultado final de esta alternativa del **case**!.

Reuniendo los resultados de las dos alternativas y eliminando variables ligadas en las guardas tenemos que el resultado final es:

$$(\{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\}, \{ys\})$$

A partir del cual extraemos la signatura $(\{2\}, \{2\}, \{2\}, \{2\}, \{2\}, \{2\}, \{2\})$. Como esta signatura difiere de la inicial, no hemos alcanzado aún un punto fijo. Actualizamos el entorno con esta información, obteniendo $\rho_1 = [revauxD \mapsto (\{2\}, \ldots, \{2\})]$ y volvemos a calcular $S\ [\![e]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho_1$. Los pasos a seguir son los mismos

que en la primera iteración, salvo en la interpretación de la llamada a *revauxD* con el nuevo entorno ρ_1, que ofrece como resultado:

$$(\{x_1\}, \{ys, x, x_1\}, \{x_1\}, \{x_1, ys\}, \{x_1, xs, x, ys\}, \{x_1, x, ys\}, \{x_1, xs, x, ys\})$$

Eliminando las variables ligadas x_1, x y xs de todos estos conjuntos obtendremos como resultado la interpretación S de la alternativa del **case**! guardada por $(x : xx)$. Reuniendo las dos alternativas se obtiene:

$$(\{ys\}, \{ys\}, \{ys\}, \{ys\}, \{xs, ys\}, \{ys\}, \{xs, ys\})$$

Con lo que resulta la signatura $(\{2\}, \{2\}, \{2\}, \{2\}, \{1, 2\}, \{2\}, \{1, 2\})$. Realizamos una tercera iteración con el entorno ρ_2, proveniente de asociar *revauxD* con esta nueva signatura. De nuevo, el cálculo de es idéntico hasta llegar a la llamada recursiva, donde su interpretación correspondiente resulta:

$$(\{x_1\}, \{ys, x, \mathbf{xs}, x_1\}, \{x_1\}, \{x_1, ys\}, \{x_1, xs, x, ys\}, \{x_1, x, ys\}, \{x_1, xs, x, ys\})$$

El aumento del segundo conjunto redunda en una modificación de la signatura, que pasa a tener $\{\mathbf{1}, 2\}$ como segundo conjunto. Sea ρ_4 el nuevo entorno modificado de este modo. Volviendo a repetir el análisis ρ_4 obtenemos el mismo resultado, por lo que se ha alcanzado el punto fijo. La signatura final para la función *revauxD* queda de la siguiente forma:

$$\boxed{revauxD \mapsto (\{2\}, \{1, 2\}, \{2\}, \{2\}, \{1, 2\}, \{2\}, \{1, 2\})}$$

El primer conjunto indica que el resultado es hijo del segundo parámetro. En efecto, esto ocurre al devolverse *ys* en el caso base de la función. Recíprocamente, esto provoca que el segundo parámetro sea también hijo del resultado, tal como muestra el cuarto conjunto. El último conjunto indica que los elementos de la lista inicial son compartidos por la lista resultado.

Ahora pasamos a la definición de *revD*. Si e es la expresión del lado derecho de esta definición, el objetivo es calcular $S\ [\![e]\!]\ SubR_0\ ShR_0\ Sub_0\ Sh_0\ \rho_4$ donde:

$$\begin{aligned} SubR_0 &= \{xs \mapsto \{xs\}\} \\ ShR_0 &= \{xs \mapsto \{xs\}\} \\ Sub_0 &= \{xs \mapsto \{xs\}\} \\ Sh_0 &= \{\{xs\}\} \end{aligned}$$

En este caso la construcción de la lista vacía no produce compartición con ninguna variable, por lo que la interpretación S para la expresión auxiliar del **let** son siete conjuntos vacíos. Esto provoca que no sea necesario acumular ninguna información en las relaciones para analizar la expresión principal. El cálculo de la misma produce como resultado:

$$(\{x_1\}, \{x_1, xs\}, \{x_1\}, \{x_1\}, \{x_1, xs\}, \{x_1\}, \{x_1, xs\})$$

Tras terminar el análisis eliminando la variable ligada x_1, la signatura correspondiente es $(\emptyset, \{1\}, \emptyset, \emptyset, \{1\}, \emptyset, \{1\})$. Si actualizamos el entorno asociando la función

revD con esta signatura (dando lugar a un entorno ρ_5) y volvemos a repetir el análisis obtendremos el mismo resultado, ya que al tratarse de una función no recursiva, el valor de $\rho_5(revD)$ no influye en el análisis. Hemos, por tanto, alcanzado el punto fijo y la signatura resultante es:

$$\boxed{revD \mapsto (\emptyset, \{1\}, \emptyset, \emptyset, \{1\}, \emptyset, \{1\})}$$

Como último ejemplo se mostrará la signatura de la función *mergeD*, que mezcla dos listas ordenadas obteniendo como resultado otra lista ordenada. El código de esta función se muestra a continuación:

```
mergeD xs ys @r =
        case! xs of
              [ ] → ys
              (x : xx) → case! ys of
                               [ ] → (x : xx)@r
                               (y : yy) → case x ≤ y of
                                                  True → (x : (mergeD xx (y : yy)@r)@r)@r
                                                  False → (y : (mergeD (x : xx)@r yy)@r)@r
```

En esta ocasion, y por motivos de claridad, se han incluido directamente expresiones no básicas en el discriminante del tercer **case** y en las construcciones y llamadas recursivas del mismo. En la sintaxis de *Core-Safe* estas expresiones habrían de ser incorporadas mediante estructuras **let**.

El análisis completo de la expresión no se muestra aquí, por motivos de espacio. No obstante ofreceremos una interpretación del resultado devuelto por el mismo. La signatura obtenida para esta definición es:

$$\boxed{mergeD \mapsto (\{2\}, \{1,2\}, \{2\}, \{2\}, \{1,2\}, \{2\}, \{1,2\})}$$

En una llamada a *mergeD* cabe la posibilidad de devolver la lista pasada como segundo parámetro. En ese caso el resultado y el segundo parámetro son la misma estructura y, por tanto, el resultado es hijo recursivo del segundo parámetro y viceversa, dada la reflexividad de la relación $\lessdot\!\!\sim$. Esto explica el significado del primer y cuarto conjuntos de la signatura. Como la relación de hijo recursivo implica a todas las demás, se tiene también el mismo 2 en el resto de conjuntos. Sin embargo, *no* ocurre lo mismo con el primer parámetro, ya que en el caso en el que la segunda lista quede vacía (primera rama de **case**! *ys* **of** . . .), se vuelve a reconstruir la primera lista a partir de la cabeza y su cola, en lugar de devolver la lista original.

Por otro lado podemos observar que aparece el primer parámetro en el segundo conjunto de la signatura. Es decir, el resultado podría apuntar a un hijo recursivo del primer parámetro. De hecho, puede verse en la segunda rama del **case**! *ys* **of** . . . cómo se devuelve una estructura a partir de *xx*, que precisamente es hijo recursivo del primer parámetro. Además, la lista *xx* es hija recursiva del resultado, por lo que el primer

parámetro también apunta a un hijo recursivo del resultado. Esto explica el hecho de que aparezca un 1 en el quinto conjunto de la signatura, a la vez que en el séptimo conjunto, debido a las implicaciones entre las distintas relaciones.

Capítulo 5

Inferencia de tipos SAFE

Los análisis de tipos Hindley-Milner y de compartición presentados en los dos capítulos anteriores sirven como base para el algoritmo de inferencia presentado en este capítulo. Dicho algoritmo tendrá como objetivo garantizar que dado un programa escrito en *Core-Safe*, las destrucciones explícitas (mediante **case**!) se realizan de forma segura, es decir, que durante la ejecución del programa no se accederá a punteros descolgados en memoria.

Para formalizar este algoritmo es necesario, en primer lugar, ampliar la sintaxis de tipos presentada en el Capítulo 3 para poder distinguir entre tipos seguros, condenados y en peligro. La nueva sintaxis será expuesta en la Sección 5.1. Posteriormente se presentará un sistema de tipos (Sección 5.2) para programas *Core-Safe* que garantiza la seguridad de las destrucciones explícitas.

Una vez expuesto el sistema de tipos, se mostrará el algoritmo que a partir de un programa *Core-Safe* encuentra un tipado correcto para el mismo. Nótese que el Capítulo 3 se ocupaba de la reconstrucción del tipo Hindley-Milner de cada elemento de un programa, por lo que este capítulo se centrará exclusivamente en la inferencia de los elementos introducidos en la nueva sintaxis de tipos. Concretamente, el algoritmo se encargará de asignar una *marca* (seguro, condenado o en peligro) a cada variable del programa y comprobará la consistencia de dicha asignación.

El algoritmo presentado en este capítulo fue publicado en [MPS07].

5.1. Expresiones de tipo

En la Figura 5.1 se define la sintaxis de tipos ampliada. Pueden encontrarse varias diferencias con respecto la sintaxis de la Figura 3.1. En primer lugar, existe una nueva categoría sintáctica τ que agrupa tipos funcionales, tipos no funcionales y tipos región. Dentro de los tipos no funcionales, seguimos distinguiendo entre tipos algebraicos, variables de tipo y tipos básicos. Sin embargo, ahora los tipos algebraicos pueden ser tipos seguros s, tipos condenados d o tipos en peligro r. Estos dos últimos son caracterizados por los signos ! y #, respectivamente, en el nivel más externo. La semántica de estos tres tipos puede expresarse informalmente de la siguiente forma:

$$
\begin{array}{lcll}
\tau & \rightarrow & t & \{\text{externo}\} \\
 & & \mid r & \{\text{en peligro}\} \\
 & & \mid \sigma & \{\text{f. polimórfica}\} \\
 & & \mid \rho & \{\text{región}\} \\
t & \rightarrow & s & \{\text{seguro}\} \\
 & & \mid d & \{\text{condenado}\} \\
s & \rightarrow & T\ \overline{\rho}\ \overline{s}@\rho' & \\
 & & \mid b & \\
d & \rightarrow & T\ \overline{\rho}\ \overline{s}!@\rho' &
\end{array}
\qquad
\begin{array}{lcll}
r & \rightarrow & T\ \overline{\rho}\ \overline{s}\#@\rho' & \\
b & \rightarrow & a & \{\text{variable}\} \\
 & & \mid B & \{\text{básico}\} \\
tf & \rightarrow & \overline{t_i}^n \rightarrow \rho' \rightarrow T\ \overline{\rho}\ \overline{s}@\rho' & \\
 & & \mid \overline{t_i}^n \rightarrow s & \\
 & & \mid \overline{s_i}^n \rightarrow \rho' \rightarrow T\ \overline{\rho}\ \overline{s}@\rho' & \{\text{constructor}\} \\
\sigma & \rightarrow & \forall a.\sigma & \\
 & & \mid \forall \rho.\sigma & \\
 & & \mid tf &
\end{array}
$$

Figura 5.1: Expresiones de tipo

- **Tipos *seguros* (*s*):** Las estructuras de este tipo pueden leerse, copiarse, ser utilizadas para construir otras estructuras, así como ser devueltas como resultado. Sin embargo, no pueden ser destruidas mediante una expresión **case**! ni reutilizarse mediante el símbolo !.

- **Tipos *condenados* (*d*):** Son aquellas estructuras que están directamente implicadas en una destrucción mediante **case**!. Esto no sólo incluye la variable que aparece como discrimante en un **case**!, sino aquellas variables en los patrones de las alternativas situados en posiciones recursivas. Cuando una variable es reutilizada (mediante !) también adquiere tipo condenado.

  Las estructuras de datos condenadas pueden leerse (mediante un **case** no destructivo) o copiarse solamente antes de ser destruidas mediante el **case**!. No obstante, no pueden utilizarse para construir otra estructura de datos o devolverse como resultado.

- **Tipos *en peligro* (*r*):** Son aquellas estructuras que, pese a no aparecer directamente condenadas en un programa, apuntan a una estructura que es hijo recursivo de una condenada. Las estructuras en peligro tienen las mismas restricciones de acceso que las condenadas, ya que representan punteros potencialmente descolgados.

EJEMPLO 14. *El siguiente fragmento de definición Core-Safe*[1] *de la función f representa la situación mostrada en la Figura 5.2.*

[1]Por motivos de claridad, a lo largo de este capítulo aparecerán ejemplos escritos con una versión "ligeramente azucarada" de *Core-Safe*, permitiéndose incluir expresiones arbitrarias como parámetros de una constructora o función. En código *Core-Safe* "estricto" estas expresiones deberían ser introducidas mediante expresiones **let**

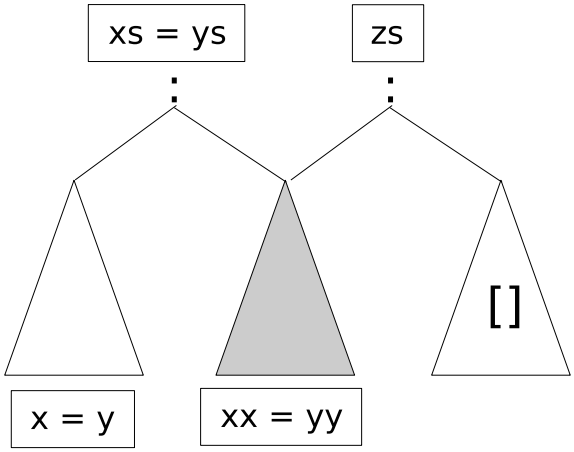

Figura 5.2: Variables condenadas y en peligro

$$
\begin{array}{l}
f\ xs\ @r = \textbf{let}\ ys = xs\ \textbf{in} \\
\qquad\qquad \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\quad [\,] \rightarrow \ldots \\
\qquad\qquad\quad (x : xx) \rightarrow \textbf{let}\ zs\ = (xx : [\,])@r\ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{case!}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \rightarrow \ldots \\
\qquad\qquad\qquad\qquad\qquad (y : yy) \rightarrow \ldots
\end{array}
$$

La variable ys aparece como condenada, así como su descendiente recursivo yy. Ambas tienen tipo $[a]!@\rho$*. Por su parte, zs apunta a un descendiente de ys, por lo que queda en peligro y su tipo es* $[a]\#@\rho$*. Del mismo modo, las variables xs y xx apuntan a una estructura condenada, por lo que también quedan en peligro. Ninguna de estas variables podrá ser devuelta como resultado en las alternativas de los* **case**.

Con respecto a los tipos funcionales *tf* conviene destacar que admiten parámetros condenados y seguros, pero no en peligro. El hecho de permitir en una función un parámetro con tipo en peligro implicaría considerar todas las variables que compartan un hijo con dicho parámetro en cada llamada a la función. Esto haría la implementación mucho más costosa, pese a no aumentar significativamente el número de programas que serían admitidos por el algoritmo. Obviamente, el tipo devuelto por una función ha de ser seguro; no tiene sentido devolver un puntero a una estructura que va a ser o ha sido destruida.

A lo largo de este capítulo no se tendrán en cuenta los tipos restringidos con desigualdades entre variables de región, por motivos de claridad. Las desigualdades obtenidas tras el algoritmo de inferencia Hindley-Milner pueden adjuntarse al tipo obtenido tras realizar la inferencia de tipos seguros.

Definimos el predicado $utype?(t, t')$, que indica si dos tipos tienen el mismo tipo subyacente (es decir, ignorando marcas ! y #). Además, en el sistema de tipos se hará uso de los siguientes predicados descritos aquí informalmente:

$$\begin{array}{rcl} safe?(\tau) & \equiv & \tau \text{ es un tipo seguro.} \\ cdm?(\tau) & \equiv & \tau \text{ es un tipo condenado.} \\ dgr?(\tau) & \equiv & \tau \text{ es un tipo en peligro.} \\ unsafe?(\tau) & \equiv & \tau \text{ es un tipo condenado o en peligro.} \\ region?(\tau) & \equiv & \tau \text{ es un tipo región.} \\ function?(\tau) & \equiv & \tau \text{ es un tipo funcional.} \end{array}$$

Por otro lado se define el siguiente orden parcial entre tipos:

$$\begin{array}{rl} \tau_1 \geq \tau_2 \iff_{def} & \tau_1 = \tau_2 \vee \\ & \tau_1 \text{ es de la forma } T\ \overline{\rho}\ \overline{s}!@\rho' \text{ y } \tau_2 \text{ es de la forma } T\ \overline{\rho}\ \overline{s}@\rho' \vee \\ & \tau_1 \text{ es de la forma } T\ \overline{\rho}\ \overline{s}\#@\rho' \text{ y } \tau_2 \text{ es de la forma } T\ \overline{\rho}\ \overline{s}@\rho' \end{array}$$

Este orden será posteriormente extendido a entornos de tipo y permitirá leer o copiar una estructura de datos condenada o en peligro antes de ser destruida.

5.2. Sistema de tipos

El sistema de tipos polimórfico descrito en esta sección permite demostrar que un programa utiliza de modo seguro las funciones de liberación de memoria del lenguaje. En las reglas de este sistema de tipos se encontrarán juicios de la forma $\Gamma^P \vdash e : \tau$ indicando que bajo cierto entorno Γ, en el contexto de una definición de función cuyos parámetros de entrada son P, la expresión e tiene tipo τ.

Dentro de un entorno Γ podemos asociar variables con su tipo $[x : \tau]$, variables de región con su tipo $[r : \rho]$, y funciones con su tipo polimórfico $[f : \sigma]$. Se utilizarán los siguientes operadores de unión entre entornos:

- **Operador** $+$: Permite la unión de dos entornos de dominios disjuntos.

$$def(\Gamma_1^P + \Gamma_2^P) \equiv dom(\Gamma_1^P) \cap dom(\Gamma_2^P) = \emptyset$$

$$\forall x \in dom(\Gamma_1^P) \cup dom(\Gamma_2^P) :$$

$$(\Gamma_1^P + \Gamma_2^P)(x) = \begin{cases} \Gamma_1^P(x) & \text{si } x \in dom(\Gamma_1^P) \\ \Gamma_2^P(x) & \text{e.o.c.} \end{cases}$$

- **Operador** $\otimes$: Permite que los dos entornos tengan variables en común, pero han

de tener el mismo tipo asociado.

$$def(\Gamma_1^P \otimes \Gamma_2^P) \equiv \forall x \in dom(\Gamma_1^P) \cap dom(\Gamma_2^P)\ .\ \Gamma_1^P(x) = \Gamma_2^P(x)$$

$$\forall x \in dom(\Gamma_1^P) \cup dom(\Gamma_2^P) :$$

$$(\Gamma_1^P \otimes \Gamma_2^P)(x) = \begin{cases} \Gamma_1^P(x) & \text{si } x \in dom(\Gamma_1^P) \\ \Gamma_2^P(x) & \text{e.o.c.} \end{cases}$$

- **Operador** $\oplus$: Se permiten variables comunes entre los dos entornos, siempre que tengan el mismo tipo subyacente en cada entorno y en ambos entornos sea un tipo seguro.

$$def(\Gamma_1^P \oplus \Gamma_2^P) \equiv \forall x \in dom(\Gamma_1^P) \cap dom(\Gamma_2^P)\ .\quad utype?(\Gamma_1^P(x), \Gamma_2^P(x)) \wedge (safe?(\Gamma_1^P(x)) \wedge safe?(\Gamma_2^P(x)))$$

$$\forall x \in dom(\Gamma_1^P) \cup dom(\Gamma_2^P) :$$

$$(\Gamma_1^P \oplus \Gamma_2^P\ (x)) = \begin{cases} \Gamma_1^P(x) & \text{si } x \in dom(\Gamma_1^P)\ \wedge\ (x \notin dom(\Gamma_2^P)\ \vee\ unsafe?(\Gamma_1^P(x))) \\ \Gamma_2^P(x) & \text{e.o.c.} \end{cases}$$

Mención aparte merece el operador $\triangleright$, cuyo uso llevará asociado dos conjuntos de variables L y C. En una unión $\Gamma_1^P \triangleright_C^L \Gamma_2^P$ no se permite que las variables de L tengan tipo inseguro en Γ_1^P, ni que las variables comunes de C y L tengan tipo inseguro en Γ_2^P. Este operador será de utilidad en las reglas de tipado para indicar que una variable no puede nombrarse tras ser destruida. En el entorno resultante de esta operación prevalecerán los tipos inseguros sobre los seguros. Su definición formal se muestra a continuación:

$$\begin{aligned} def(\Gamma_1^P \triangleright_C^L \Gamma_2^P) \equiv\ & (\forall x \in dom(\Gamma_1^P) \cap dom(\Gamma_2^P).\ utype?(\Gamma_1^P(x), \Gamma_2^P(x))) \wedge \\ & (\forall x \in dom(\Gamma_1^P).\ unsafe?(\Gamma_1^P(x)) \rightarrow x \notin L) \wedge \\ & (\forall x \in C \cap L \cap dom(\Gamma_2^P).\neg unsafe?(\Gamma_2^P(x))) \end{aligned}$$

$$\forall x \in dom(\Gamma_1^P) \cup dom(\Gamma_2^P) :$$

$$\Gamma_1^P \triangleright_C^L \Gamma_2^P(x) = \begin{cases} \Gamma_2^P(x) & \text{si } x \notin dom(\Gamma_1^P) \vee \\ & \quad (x \in dom(\Gamma_1^P) \cap dom(\Gamma_2^P) \wedge safe(\Gamma_1^P(x))) \\ \Gamma_1^P(x) & \text{e. o. c.} \end{cases}$$

En la Figura 5.3 se muestran las reglas de tipo para expresiones. Existe un invariante en el sistema de tipos que garantiza que si una variable aparece condenada en un entorno de tipos, aquellas variables que compartan un hijo recursivo suyo deberán

$$\frac{\Gamma^P \vdash e : s \quad x \notin dom(\Gamma^P) \quad safe?(\tau) \vee danger?(\tau) \vee region?(\tau) \vee function?(\tau)}{\Gamma^P + [x : \tau] \vdash e : s}\ [\text{EXTS}] \qquad \frac{\Gamma^P \vdash e : s \quad x \notin dom(\Gamma^P) \quad R = sharerec(x, e) - \{x\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}}{\Gamma^P \otimes \Gamma_R + [x : d] \vdash e : s}\ [\text{EXTD}]$$

$$\frac{}{\emptyset \vdash c : B}\ [\text{LIT}] \qquad \frac{}{[x : s]^P \vdash x : s}\ [\text{VAR}] \qquad \frac{}{[r : \rho]^P \vdash r : \rho}\ [\text{REGION}] \qquad \frac{tf \trianglelefteq \sigma}{[f : \sigma]^P \vdash f : tf}\ [\text{FUNCTION}]$$

$$\frac{R = sharerec(x, x!) - \{x\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}}{\Gamma_R + [x : T!@\rho]^P \vdash x! : T@\rho}\ [\text{REUSE}] \qquad \frac{\Gamma_1^P \geq_{x@r} [x : T@\rho', r : \rho] \quad \rho \neq \rho'}{\Gamma_1^P \vdash x@r : T\ @\rho}\ [\text{COPY}]$$

$$\frac{\Gamma_1^P \vdash e_1 : s_1 \quad \Gamma_2^P + [x_1 : s_1] \vdash e : s \quad C = shareall(x_1, e)}{\Gamma_1^P \rhd_C^{fv(e)} \Gamma_2^P \vdash \textbf{let}\ x_1 = e_1\ \textbf{in}\ e : s}\ [\text{LET1}]$$

$$\frac{\Gamma_1^P \vdash e_1 : s_1 \qquad \Gamma_2^P + [x_1 : d_1] \vdash e : s \qquad d_1 = \overline{s_1} \quad C = shareall(x_1, e) \qquad R = sharerec(x_1, e) \qquad P \cap R = \emptyset}{\Gamma_1^P \rhd_{C-R}^{fv(e)} \Gamma_2^P \vdash \textbf{let}\ x_1 = e_1\ \textbf{in}\ e : s}\ [\text{LET2}]$$

$$\frac{\overline{t_i}^n \to \rho \to T\ @\rho \trianglelefteq \sigma \qquad \Gamma^P = [f : \sigma] + [r : \rho] + \bigoplus_{i=1}^n [a_i : t_i] \quad R = \bigcup_{i=1}^n \{sharerec(a_i, (f\ \overline{a_i}^n)@r) - \{a_i\} \mid cdm?(t_i)\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}}{\Gamma_R + \Gamma^P \vdash f\ \overline{a_i}^n\ @r : T\ @\rho}\ [\text{APP1}]$$

$$\frac{\overline{t_i}^n \to s \trianglelefteq \sigma \qquad \Gamma^P = [f : \sigma] + \bigoplus_{i=1}^n [a_i : t_i] \quad R = \bigcup_{i=1}^n \{sharerec(a_i, (f\ \overline{a_i}^n)@r) - \{a_i\} \mid cdm?(t_i)\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}}{\Gamma_R + \Gamma^P \vdash f\ \overline{a_i}^n\ @r : s}\ [\text{APP2}]$$

$$\frac{\Sigma(C) = \sigma \qquad \overline{s_i}^n \to \rho \to T\ @\rho \trianglelefteq \sigma \qquad \Gamma^P = \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma^P \vdash (C\ \overline{a_i}^n)@r : T\ @\rho}\ [\text{CONS}]$$

$$\frac{(\forall i \in \{1..n\}).\Sigma(C_i) = \sigma_i \qquad (\forall i \in \{1..n\}).\overline{s_{ij}}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \quad \Gamma^P \geq_{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n} [x : T@\rho] \qquad (\forall i \in \{1..n\}).\forall j \in \{1..n_i\}).inh(\tau_{ij}, s_{ij}, \Gamma^P(x)) \quad (\forall i \in \{1..n\}).\Gamma^P + \overline{[x_{ij} : \tau_{ij}]}^{n_i} \vdash e_i : s}{\Gamma^P \vdash \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE}]$$

$$\frac{\begin{array}{ll}(\forall i \in \{1..n\}).\Sigma(C_i) = \sigma_i & (\forall i \in \{1..n\}).\overline{s_{ij}}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \\ R = sharerec(x, \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) - \{x\} & (\forall i \in \{1..n\}.\forall j \in \{1..n_i\}).inh!(t_{ij}, s_{ij}, T\ !@\rho) \\ \forall z \in R \cup \{x\}, i \in \{1..n\}.z \notin fv(e_i) & (\forall i \in \{1..n\}).\Gamma^P + [x : T\ \#@\rho] + \overline{[x_{ij} : t_{ij}]}^{n_i} \vdash e_i : s \\ \multicolumn{2}{c}{\Gamma_R = \{y : danger(type(y)) \mid y \in R\}}\end{array}}{\Gamma_R \otimes \Gamma^P + [x : T\ !@\rho] \vdash \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE!}]$$

Figura 5.3: Reglas de tipado para expresiones

aparecer también en el entorno con tipo inseguro. Este invariante (demostrado en la Sección 6.2) resulta necesario para propagar la información de punteros posiblemente descolgados (variables en peligro) desde la expresión auxiliar de un **let** a la expresión principal del mismo.

La regla [LIT] asocia a cada literal su tipo básico correspondiente. Las reglas [VAR], [REGION] y [FUNCTION] asignan tipo a las variables, obteniendo la información pertinente del entorno de tipos. En esta última regla se utiliza la notación $tf \trianglelefteq \sigma$ para indicar la instanciación (con tipos seguros) de un tipo polimórfico.

Estas cuatro reglas sólo son aplicables bajo un entorno mínimo. Si se quiere ampliar el entorno con nuevas variables puede hacerse mediante las reglas [EXTS] y [EXTD] que permiten la ampliación restringida de un entorno atendiendo a las siguientes reglas:

1. Sólo pueden añadirse variables frescas.

2. La regla [EXTS] permite añadir variables con tipos seguros, en peligro, región o funcionales.

3. La regla [EXTD] permite añadir una variable con tipo condenado. No obstante, todas las variables que compartan un hijo recursivo suyo (ver Sección 4.5.5 para definición de *sharerec*) deberán ser añadidos al entorno con tipo en peligro. Esto se hace para preservar el invariante del sistema de tipos. La notación $danger(t)$ obtiene la versión en peligro del tipo Hindley-Milner t.

Para poder tipar una expresión de la forma $x!$ (regla [REUSE]) es necesario que aparezca con tipo condenado en el entorno y, al igual que en [EXTD], que todas las variables que compartan un hijo recursivo suyo aparezcan en peligro.

Mediante la regla [COPY] se indica que puede copiarse una variable de cualquier tipo no básico. El operador $\geq$ definido sobre entornos permite que pueda copiarse una variable condenada antes de ser destruida. Este operador se define en el ámbito de una expresión e de la siguiente forma:

$$\begin{aligned}\Gamma_1 \geq_e \Gamma_2 \equiv\ & dom(\Gamma_2) \subseteq dom(\Gamma_1) \wedge \\ & \forall x \in dom(\Gamma_2).\Gamma_1(x) \geq \Gamma_2(x) \wedge \\ & \forall x \in dom(\Gamma_1).cmd?(\Gamma_1(x)) \rightarrow \forall z \in sharerec(x,e).z \in dom(\Gamma_1) \ \wedge \\ & \qquad\qquad\qquad\qquad\qquad\qquad unsafe?(\Gamma_1(z))\end{aligned}$$

Para preservar el invariante del sistema de tipos se asegura que si en Γ_1 se añade una variable con tipo condenado que no aparezca en Γ_2 se añaden también las variables que compartan un hijo recursivo de la misma.

Para poder asignar tipo a una expresión **let** distinguimos dos casos según si la variable que se está definiendo se destruya o no en la expresión principal del mismo. La regla [LET1] supone que la variable x_1 no se destruye ni queda en peligro en la expresión e. En este caso el operador $\lhd$ garantiza que:

1. Ninguna variable que se destruya o quede en peligro en e_1 puede aparecer libre en e, ya que representa un puntero descolgado. (La función *fv* devuelve las variables libres de una expresión).

2. No puede utilizarse destructivamente en e cualquier variable que comparta algo con x_1, ya que en ese caso la estructura de x_1 recién construida quedaría corrupta.

En la regla [LET2] se permite que la variable x_1 pueda destruirse en e. Utilizamos la notación $\overline{s}$ para indicar la versión condenada del tipo seguro s. En este caso el operador $\lhd$ asegura que:

1. Al igual que en la regla [LET1], ninguna variable que se destruya o quede en peligro en e_1 puede aparecer libre en e.

2. No puede utilizarse destructivamente en e cualquier variable que comparta un hijo *no recursivo* de x_1. Las subestructuras recursivas de x_1 sí pueden ser eliminadas.

3. Ningún parámetro de entrada a la función puede apuntar a un hijo recursivo de x_1, ya que la estructura que representa quedaría potencialmente corrupta al destruir x_1.

La regla [APP1] se encarga de la aplicación de una función. En este caso se comprueba que los parámetros y la región de salida tengan un tipo acorde con la signatura de la misma. El operador $\oplus$ definido anteriormente impide que una misma variable pueda aparecer en distintas posiciones condenadas de la función. Al igual que en reglas anteriores se añade un entorno Γ_R que contiene las variables que pueden quedar en peligro tras la llamada, preservando así el invariante.

La construcción de estructuras de datos (regla [CONS]) es similar a la anterior, pero más restrictiva, ya que todos los argumentos han de tener tipo seguro. En efecto, no tendría sentido construir una estructura de datos con valores potencialmente destruíbles. Por otro lado, suponemos contenidas en un entorno Σ los esquemas de tipo de cada constructora del programa.

La regla [CASE] permite que el discriminante de un **case** pueda leerse antes de ser destruído. Para expresar esto se utiliza el orden $\geq$ definido sobre entornos visto anteriormente. El tipado de cada alternativa plantea una dificultad adicional, ya que aparecen nuevas variables (patrones de cada alternativa) que han de llevar un tipo asignado. Para ello definimos la relación *inh*, que determina los tipos *heredados* por las variables que aparecen en el patrón de cada alternativa:

$$\begin{array}{lcl} inh(s, s, \tau) & \Leftrightarrow & safe?(\tau) \vee dgr?(\tau) \vee (\neg utype?(s, \tau) \wedge cmd?(\tau)) \\ inh(danger(s), s, \tau) & \Leftrightarrow & dgr?(\tau) \vee (utype?(s, \tau) \wedge cmd?(\tau)) \end{array}$$

De este modo:

1. Si el discriminante del **case** tiene tipo **seguro**, también lo tendrán las variables patrón correspondientes.

2. Si el discriminante del **case** tiene tipo **en peligro**, las variables patrón correspondientes pueden tener tipo seguro o en peligro.

3. Si el discriminante del **case** tiene tipo **condenado**, las variables que ocupan posiciones recursivas en el patrón de la alternativa tendrán tipo en peligro, mientras que que el resto tendrán tipo seguro.

En un **case** destructivo (regla [CASE!]), el discriminante x está siendo destruido, por lo que no podrá ser nombrado en las alternativas. Del mismo modo reunimos todas las variables que comparten un hijo recursivo de x en el conjunto R. Estas variables tampoco podrán ser nombradas en las alternativas. La relación *inh*! adjudica un tipo heredado a cada variable del patrón de cada alternativa:

$$\begin{array}{lcl} inh!(s,s,d) & \Leftrightarrow & \neg utype?(s,d) \\ inh!(d,s,d) & \Leftrightarrow & utype?(s,d) \end{array}$$

En este caso el discriminante x siempre tiene tipo condenado, así como las variables patrón que estén en posiciones recursivas del constructor correspondiente. El resto de variables patrón adquieren tipo seguro (sólo lectura).

5.3. Algoritmo de inferencia

Una vez introducidas las reglas del sistema de tipos podemos describir el algoritmo que determina el tipo de cada elemento del árbol abstracto correspondiente a un programa. En primer lugar es necesario destacar que las reglas de la Figura 5.3 permiten en principio varios tipados correctos para un programa.

EJEMPLO 15. *Se define la función join, que concatena dos listas:*

$$\begin{array}{l} join\ xs\ ys\ @r = \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad\qquad\qquad [\,] \rightarrow ys \\ \qquad\qquad\qquad\qquad (x : xx) \rightarrow (x : join\ xx\ ys\ @r)@r \end{array}$$

El sistema de tipos admite varios tipados para la función join, entre ellos:

1. $join :: [a]@\rho' \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho$

2. $join :: [a]!@\rho' \rightarrow [a]@\rho \rightarrow \rho \rightarrow [a]@\rho$

El segundo tipo especifica que la lista pasada como primer parámetro podría destruirse durante la llamada a join. Sin embargo, según la definición de la función dicha lista nunca es destruida. En un contexto donde se realice una llamada a la función join la variable que ocupe el primer parámetro sería marcada como condenada, lo cual impediría que fuese reutilizada, por ejemplo, para ser devuelta como parte del resultado. Esta restricción innecesaria nos lleva a considerar el tipo 1 como el más adecuado para la función join.

Tal como muestra el ejemplo, el sistema de tipos puede asignar tipos condenados y en peligro, aunque no sea necesario. El objetivo es obtener tipos *minimos*, en el sentido de ser lo más polimórficos posible y poseer la menor cantidad de tipos inseguros posible. La inferencia de tipos Hindley-Milner vista en el Capítulo 3 ya se encarga de calcular el tipo más polimórfico posible, por lo que el algoritmo de inferencia presentado a continuación se encargará de satisfacer la segunda propiedad.

El algoritmo consiste en un recorrido del árbol abstracto, que se realiza en dos sentidos. En principio el árbol abstracto se recorre *bottom-up*, es decir, desde las hojas hasta la raíz. Este proceso puede ser interrumpido a lo largo del análisis para realizar ciertas comprobaciones de coherencia entre marcas asignadas a las variables. Una comprobación de este tipo recibe el nombre de **check** y se realiza en sentido *top-down* (descendente) desde el punto en que el recorrido *bottom-up* fue interrumpido. Si tiene éxito, se procederá a reanudar el recorrido *bottom-up*.

5.3.1. Recorrido *bottom-up* del árbol abstracto

Durante el recorrido ascendente del árbol abstracto se van acumulando conjuntos que especifican la marca de las variables. Los conjuntos propagados son cuatro:

- Conjunto D de variables condenadas.
- Conjunto R de variables en peligro.
- Conjunto S de variables seguras.
- Conjunto N de variables cuya marca es temporalmente desconocida.

El último de estos conjuntos surge a partir del hecho de que puede realizarse una copia o un **case** no destructivo sobre una variable condenada, en peligro o segura. Las variables que pertenezcan al conjunto N podrían ser transferidas a cualquiera de los tres conjuntos restantes en un contexto superior.

En la Figura 5.4 se definen las reglas que definen el recorrido *bottom-up* del árbol abstracto. Un juicio de la forma $e \vdash_{inf} (D, R, S, N)$ indica que a partir de la expresión e se obtienen los conjuntos de marcas D, R, S y N. De este modo, colocando las marcas a los tipos correspondientes puede construirse un entorno con el que tipar la expresión. Además consideramos que dicha expresión e queda *decorada* con estos cuatro conjuntos de marcas. Esta decoración será posteriormente utilizada en el recorrido *top-down*.

A partir de este conjunto de reglas puede deducirse el invariante de que los conjuntos D, R, S y N inferidos a partir de una expresión e son disjuntos dos a dos y que toda variable libre de e ha de encontrarse en uno de estos conjuntos (ver Sección 6.2). El conjunto R podría además contener variables que no estén libres en e, pero sí en ámbito. Esto último proviene del uso de las funciones *sharerec* y *shareall*, que devuelven las variables que cumplen la propiedad de compartición correspondiente de entre todas las variables en ámbito.

$$\frac{}{c \vdash_{inf} (\emptyset,\emptyset,\emptyset,\emptyset)}\ [\mathrm{LIT}_I] \qquad \frac{}{x \vdash_{inf} (\emptyset,\emptyset,\{x\},\emptyset)}\ [\mathrm{VAR}_I] \qquad \frac{}{x@r \vdash_{inf} (\emptyset,\emptyset,\emptyset,\{x\})}\ [\mathrm{COPY}_I]$$

$$\frac{R = \mathit{sharerec}(x, x!) - \{x\} \quad \mathit{type}(x) = T@\rho}{x! \vdash_{inf} (\{x\}, R, \emptyset, \emptyset)}\ [\mathrm{REUSE}_I] \qquad \frac{\forall i \in \{1..n\}.a_i \vdash_{inf} (\emptyset,\emptyset,S_i,\emptyset)}{\overline{Ca_i}^n@r \vdash_{inf} (\emptyset,\emptyset,\bigcup_{i=1}^n S_i,\emptyset)}\ [\mathrm{CONS}_I]$$

$$\frac{\begin{array}{lll}\forall i \in \{1..n\}.D_i = \{a_i \mid i \in I_D\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n S_i) = \emptyset & R \cap (\bigcup_{i=1}^n S_i) = \emptyset \\ \forall i \in \{1..n\}.S_i = \{a_i \mid i \in I_S\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n N_i) = \emptyset & R \cap (\bigcup_{i=1}^n D_i) = \emptyset \\ \forall i \in \{1..n\}.N_i = \{a_i \mid i \in I_N\} & \forall i,j \in \{1..n\}\,.\, i \neq j \Rightarrow D_i \cap D_j = \emptyset & R \cap (\bigcup_{i=1}^n N_i) = \emptyset \\ \Sigma \vdash f : (I_D,\emptyset,I_S,I_N) & R = \bigcup_{i=1}^n \{\mathit{sharerec}(a_i, f\overline{a_i}^n@r) - \{a_i\} \mid a_i \in D_i\} & \end{array}}{f\overline{a_i}^n@r \vdash_{inf} (\bigcup_{i=1}^n D_i, R, \bigcup_{i=1}^n S_i, (\bigcup_{i=1}^n N_i) - (\bigcup_{i=1}^n S_i))}\ [\mathrm{APP}_I]$$

$$\frac{\begin{array}{rl} C = \begin{cases} \mathit{shareall}(x_1,e_2) & \text{si } x_1 \notin D_2 \cup R_2 \\ \mathit{shareall}(x_1,e_2) - \mathit{sharerec}(x_1,e_2) & \text{si } x_1 \in D_2 \end{cases} & \\ e_1 \vdash_{inf} (D_1,R_1,S_1,N_1) & N = (N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2 \\ e_2 \vdash_{inf} (D_2,R_2,S_2,N_2) & \mathit{def}((D_1 \cup R_1) \rhd_C^{FV(e_2)} (D_2 \cup R_2)) \\ (\emptyset,\emptyset,N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{\mathit{check}} e_1 & (\emptyset,\emptyset,(S_1 \cup \{x_1\}) \cap N_2) \vdash_{\mathit{check}} e_2 \end{array}}{\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \vdash_{inf} ((D_1 \cup D_2) - \{x_1\}, R_1 \cup (R_2 - D_1), ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2), N - \{x_1\})}\ [\mathrm{LET}_I]$$

$$\frac{\begin{array}{l} \forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i,R_i,S_i,N_i) \qquad \mathit{def}(\sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad (D,R,S,N) = \sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} \quad \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}(\mathit{type}(x), D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \mathit{type}(x) = \begin{cases} d & \text{si } x \in D \\ r & \text{si } x \in R \\ s & \text{si } x \in S \\ n & \text{e. o. c.} \end{cases} \quad N' = \begin{cases} N & \text{si } x \in D \cup R \cup S \\ N \cup \{x\} & \text{si } x \notin D \cup R \cup S \end{cases} \\ \forall i \in \{1..n\}\,.\, R_i'' = \{y \in P_i \cap \mathit{sharerec}(z,e_i) \mid z \in (D \cup D_i') \cap N_i\} \\ \forall i \in \{1..n\}\,.\, R_i''' = \{y \in D \cap \mathit{sharerec}(z,e_i) \mid z \in (D \cup D_i') \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\, R_i'' \cap (S_i \cup S_i') = \emptyset \\ \forall i \in \{1..n\}\,.\, ((D \cup D_i') \cap N_i, R \cup ((R_i' \cup R_i'' \cup R_i''') - D_i), (S \cup S_i') \cap N_i) \vdash_{\mathit{check}} e_i \\ \mathbf{donde}\ D_i' = \emptyset \quad R_i' = \begin{cases} \mathit{Rec}_i & \text{si } \mathit{type}(x) = d \\ \emptyset & \text{e.o.c.} \end{cases} \quad S_i' = \begin{cases} P_i & \text{si } \mathit{type}(x) = s \\ P_i - (R_i \cup S_i) & \text{si } \mathit{type}(x) = r \\ P_i - \mathit{Rec}_i & \text{si } \mathit{type}(x) = d \\ \emptyset & \text{e.o.c.} \end{cases} \end{array}}{\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D,R,S,N')}\ [\mathrm{CASE}_I]$$

$$\frac{\begin{array}{l} \forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i,R_i,S_i,N_i) \qquad \mathit{def}(\sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad (D,R',S,N) = \sqcup_{i=1}^n (D_i,R_i,S_i,N_i,P_i) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} \quad \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}!(D_i,R_i,S_i,P_i,\mathit{Rec}_i)) \\ R = \mathit{sharerec}(x, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) - \{x\} \qquad \mathit{def}((R \cup \{x\}) \rhd_\emptyset^L (D \cup R')) \\ L = \bigcup_{i=1}^n FV(e_i) \qquad \mathit{type}(x) = T@\rho \\ \forall i \in \{1..n\}\,.\, ((D \cup \mathit{Rec}_i) \cap N_i, R' \cup ((R_i' \cup R_i'' \cup R_i''') - D_i), (S \cup (P_i - \mathit{Rec}_i)) \cap N_i) \vdash_{\mathit{check}} e_i \\ \mathbf{donde}\ R_i' = \{y \in P_i \cap \mathit{sharerec}(x,e_i) \mid x \in D \cap N_i\} - ((D \cup \mathit{Rec}_i) \cap N_i) \\ R_i'' = \{y \in \mathit{sharerec}(x,e_i) \mid\ x \in \mathit{Rec}_i \cap N_i\} - ((D \cup \mathit{Rec}_i) \cap N_i) \\ R_i''' = \{y \in D \cap \mathit{sharerec}(z,e_i) \mid z \in D \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\, R_i' \cap S_i = \emptyset \ \wedge\ R_i' \cap (P_i - \mathit{Rec}_i) = \emptyset \ \wedge\ R_i'' \cap S = \emptyset \end{array}}{\mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D \cup \{x\}, R \cup (R' - \{x\}), S, N)}\ [\mathrm{CASE!}_I]$$

Figura 5.4: Reglas de inferencia *bottom-up*

La motivación de las reglas de la Figura 5.4 está en correspondencia con la explicación de las reglas del sistema de tipos explicadas en la Sección 5.2. Se detallarán a continuación sólo aquellos aspectos en las reglas de inferencia que difieren del sistema de tipos:

En la regla [REUSE$_I$], además de incluir la variable correspondiente en el conjunto de las condenadas y su correspondiente *sharerec* en el conjunto R, se comprueba que el tipo de la variable x sea *algebraico*, ya que no tiene sentido reutilizar una variable de tipo básico cuyo contenido no se encuentra alojado en el montón.

El caso de la aplicación de una función (regla [APP$_I$]) se supone la existencia de una signatura Σ que asocia nombres de función con su signatura de destrucción correspondiente. Una signatura de destrucción es una tupla de conjuntos de enteros (I_D, I_R, I_S, I_N) que indican las posiciones de los parámetros de la función condenados, en peligro, sólo lectura o indeterminado, respectivamente. En ninguna función se permiten parámetros en peligro, por lo que el conjunto I_R siempre debería ser vacío. En el caso de funciones recursivas esta signatura aún no es conocida, por lo que será necesario partir de una signatura en la que todos los parámetros tienen marca desconocida e iterar el análisis hasta alcanzar un punto fijo.

Las reglas [LET1] y [LET2] del sistema de tipos quedan agrupadas en una regla de inferencia [LET$_I$]. El caso en que $x_1 \notin D_2 \cup R_2$ se corresponde con [LET1] y el caso en que $x_1 \in D_2$ se corresponde con [LET2]. Del mismo modo se supone definido el operador $\rhd_C^L$ sobre conjuntos, cuya definición es análoga al operador de mismo nombre aplicado a entornos en el sistema de tipos:

$$\begin{array}{rcl} def(U_1 \rhd_C^L U_2) & \equiv & U_1 \cap L = \emptyset \wedge \\ & & L \cap C \cap U_2 = \emptyset \end{array}$$

Por otro lado puede observarse en la regla [LET$_I$] la notación $(D_p, R_p, S_p) \vdash_{check} e$, que indica el recorrido *top-down* de e para comprobar la consistencia de variables que han adquirido tipo seguro en ambas subexpresiones del **let**. Esto será explicado en la Sección 5.3.2.

En la regla [CASE$_I$] se introduce el operador $\sqcup$, que se encarga de reunir las tuplas (D_i, R_i, S_i, N_i) obtenidas a partir de las alternativas de un **case** en una sóla tupla (D, R, S, N). Esta información ha de ser coherente entre las distintas alternativas del mismo. Por ejemplo, una variable no puede aparecer con tipo seguro en un alternativa (conjunto S) y con tipo condenado en otra (conjunto D). Si una misma variable ha sido marcada en dos alternativas diferentes, o bien se trata la misma marca en las dos alternativas, o bien una de esas marcas es desconocido (esto es, pertenece al conjunto N). En este último caso prevalecerá la marca que no sea desconocida, si existe. La definición de $\sqcup$ se muestra a continuación:

$$\begin{array}{rcll} def(\sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i)) & \equiv & \forall i, j \in \{1..n\} \,.\, i \neq j \Rightarrow & (D_i - P_i) \cap (R_j - P_j) = \emptyset \wedge \\ & & & (D_i - P_i) \cap (S_j - P_j) = \emptyset \wedge \\ & & & (R_i - P_i) \cap (S_j - P_j) = \emptyset \end{array}$$

$$\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \stackrel{\text{def}}{=} (D, R, S, N) \textbf{ donde } \begin{cases} D &= \bigcup_{i=1}^{n}(D_i - P_i) \\ R &= \bigcup_{i=1}^{n}(R_i - P_i) \\ S &= \bigcup_{i=1}^{n}(S_i - P_i) \\ N &= (\bigcup_{i=1}^{n}(N_i - P_i)) - (D \cup R \cup S) \end{cases}$$

Por otro lado se realiza la comprobación de las marcas heredadas por las variables patrón de cada alternativa mediante la definición de *inh*, análoga a la relación del mismo nombre en el sistema de tipos:

$$\begin{array}{lcl} def(inh(n, D_i, R_i, S_i, P_i, Rec_i)) & \equiv & true \\ def(inh(s, D_i, R_i, S_i, P_i, Rec_i)) & \equiv & P_i \cap (D_i \cup R_i) = \emptyset \\ def(inh(r, D_i, R_i, S_i, P_i, Rec_i)) & \equiv & P_i \cap D_i = \emptyset \\ def(inh(d, D_i, R_i, S_i, P_i, Rec_i)) & \equiv & Rec_i \cap (D_i \cup S_i) = \emptyset \ \wedge \ (P_i - Rec_i) \cap (D_i \cup R_i) = \emptyset \end{array}$$

La regla [CASE!$_I$] también hace uso del operador $\sqcup$ para reunir la información de las distintas alternativas. La relación *inh*! viene definida de la siguiente forma:

$$def(inh!(D_i, R_i, S_i, P_i, Rec_i)) \equiv Rec_i \cap (R_i \cup S_i) = \emptyset \ \wedge \ (P_i - Rec_i) \cap (D_i \cup R_i) = \emptyset$$

5.3.2. Recorrido *top-down* del árbol abstracto

Es posible que durante el recorrido *bottom-up* no se disponga aún de información suficiente para asignar una marca a una variable. En ese caso dicha variable quedaría marcada como desconocida. No obstante, esta misma variable podría obtener una marca diferente en un contexto superior, por lo que sería necesario volver a descender por el árbol abstracto de sintaxis para comprobar que la nueva marca puede ser propagada.

EJEMPLO 16. *La siguiente definición de función es incorrecta:*

```
f num xs @r =
    case num of
        0 → case xs of
                [ ] → [ ]@r
                (x : xx) → xx
        n → case! xs of
                ys → [ ]@r
```

En efecto, el parámetro xs se destruye en la segunda alternativa del **case** *más externo, con lo que queda marcado como condenado. Sin embargo, uno de los posibles resultados de la función es xx, que es hijo recursivo de una variable condenada y, por tanto, también adquiere tipo condenado.*

Sean (D_1, R_1, S_1, N_1) *y* (D_2, R_2, S_2, N_2) *los resultados del recorrido* bottom-up *de la primera y segunda alternativa, respectivamente, del* **case** *más externo. De la primera alternativa tenemos* $xs \in N_1$ *y de la segunda tenemos* $xs \in D_2$*. Ahora es necesario volver a descender hasta la primera rama para comprobar que xs admite tipo condenado en la misma. Esto implicaría*

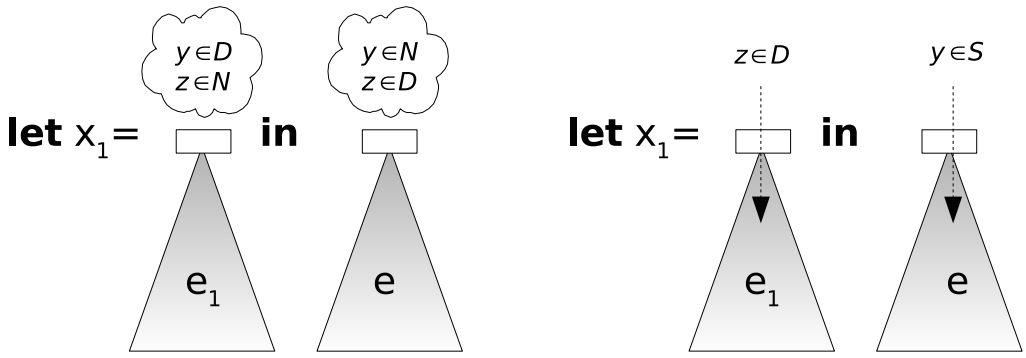

Figura 5.5: Propagación de variables en **let**

que xx también adquiriese tipo condenado, ya que aparece como hijo recursivo de xs. Sin embargo, xx fue marcada anteriormente como segura ($xx \in S_1$), ya que se devolvía como resultado de una función. Debido a este conflicto, el algoritmo de inferencia rechazará esta definición.

Durante el recorrido *bottom-up* los únicos cambios de marca que puede sufrir una variable es pasar de tener marca N a tener marca D,S ó R. Esto puede ocurrir en los siguientes tres casos:

- Una variable adquiere marca N en la expresión auxiliar (resp. principal) de un **let** y marca distinta de N en la expresión principal (resp. auxiliar). Es necesario propagar la marca distinta de N en la expresión auxiliar (resp principal). Esta situación se muestra en la Figura 5.5.

- En algunas alternativas de un **case**(!) una variable tiene una misma marca distinta de N y en el resto de alternativas dicha variable tiene marca N. Es necesario propagar la marca distinta de N en estas últimas. Esta situación se muestra en la Figura 5.6.

- Una variable que aparece como patrón de una alternativa de un **case**(!) adquiere una marca N en dicha alternativa, pero el discriminante del **case**(!) obtiene una marca distinta de N. Es necesario propagar el tipo heredado de la variable del patrón por la rama correspondiente (Figura 5.7).

Esta propagación de variables (conocida como ***check***) se realiza recorriendo el árbol abstracto en sentido *top-down*. Una vez que la propagación de variables ha tenido éxito puede reanudarse el recorrido *bottom-up*. Durante dicha propagación se comprueba que realmente es posible asignar la nueva marca a la variable propagada.

Las reglas que dirigen este recorrido *top-down* vienen dadas en la Figura 5.8. Las comprobaciones realizadas son análogas a las de las reglas de inferencia para el recorrido *bottom-up*. Mediante un juicio de la forma $(D_p, R_p, S_p) \vdash_{check} e$ indicamos que las

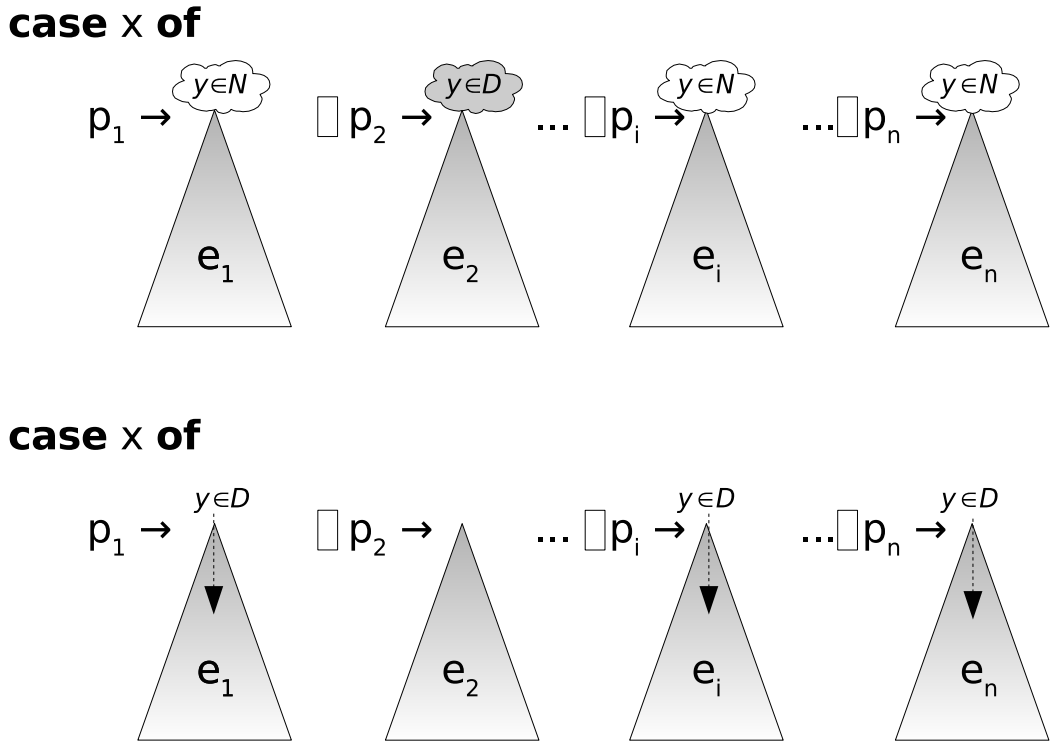

Figura 5.6: Propagación en **case** para asegurar la compatibilidad entre ramas

variables contenidas en D_p (respectivamente R_p y S_p) fueron inferidas en e con marca desconocida y que en un contexto superior se le ha asignado marca D (respectivamente R y S).

Entre las reglas de la Figura 5.8 podemos destacar [EMPTY$_C$], que da por finalizado el check en el caso en que no haya ninguna variable que propagar. En las reglas [LET$_C$], [CASE$_C$] y [CASE!$_C$] encontramos apariciones de $\vdash_{inf}$. Dichas apariciones no deben interpretarse como una nueva invocación de las reglas *bottom-up* de inferencia, sino como un acceso a los conjuntos que fueron inferidos y almacenados previamente como decoración del árbol abstracto. No hay necesidad de calcularlos de nuevo.

Aunque no se indica explícitamente en las reglas, es importante remarcar que un $\vdash_{check}$ **modifica la decoración de los elementos del árbol abstracto**, ya que algunas variables que estaban marcadas como N pasan a tener otra marca. Dada una expresión e en la que se ha inferido $e \vdash_{inf} (D, R, S, N)$ y se realiza posteriormente $(D_p, R_p, S_p) \vdash_{check} e$, la nueva decoración asociada a la subexpresión e pasa a ser:

$$(D \cup D_p, R \cup R_p, S \cup S_p, N - (D_p \cup R_p \cup S_p))$$

Aunque una misma subexpresión e puede sufrir varios $\vdash_{check}$ durante el algoritmo de inferencia siempre lo hará con un conjunto distinto de variables cada vez. Esto es

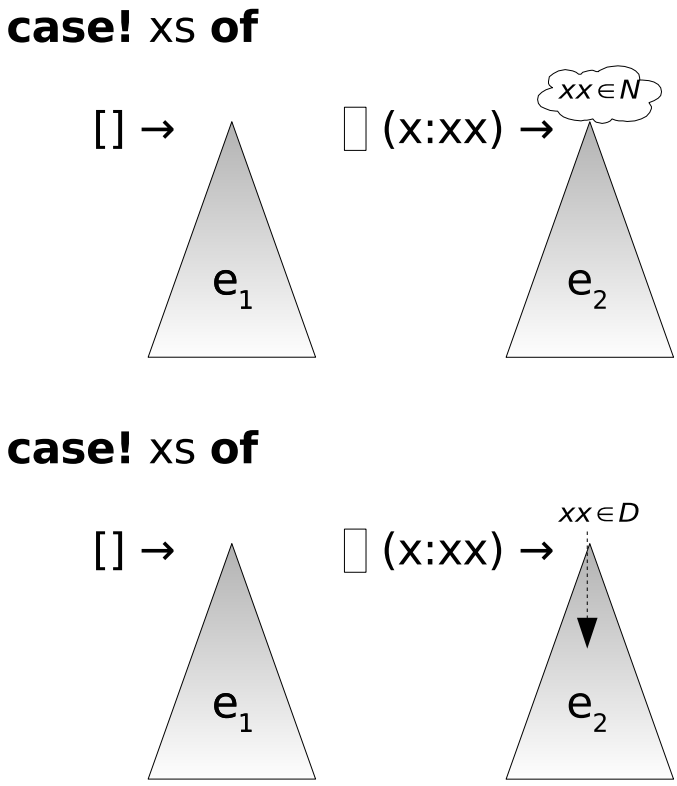

Figura 5.7: Propagación de variables en **case**! por tipos heredados

debido a que en la decoración de e se eliminan del conjunto N las variables sobre las que se ha hecho un check (esto es, $D_p \cup R_p \cup S_p$). De este modo un posible $\vdash_{check}$ posterior sobre e no podrá incluir estas variables, ya que sólo las variables N pueden cambiar de marca.

5.3.3. Comprobaciones finales

El algoritmo de inferencia puede aplicarse de forma independiente para cada definición que forme parte de un programa *Core-Safe*. Dada una definición $f\ \overline{x_i}^n\ @r = e$, el algoritmo puede haber inferido $e \vdash_{inf} (D, R, S, N)$. Esto puede haber provocado a su vez varios $\vdash_{check}$ sobre las subexpresiones de e. A partir de los conjuntos (D, R, S, N) inferidos puede obtenerse la signatura correspondiente (I_D, I_R, I_S, I_N), donde se asigna una marca a cada parámetro $(I_D, I_R, I_S, I_N \subseteq \{1..n\})$.

Como se ha dicho anteriormente, si la función f es recursiva es necesario calcular un punto fijo para hallar el tipo SAFE. La signatura inicial de f asigna la marca N a todos los parámetros. Tras cada iteración, algunos parámetros pueden haber cambiado de marca, por lo que es necesario volver a realizar $\vdash_{inf}$ sobre e hasta que la signatura se estabilice.

$$\frac{}{(\emptyset, R, \emptyset) \vdash_{check} c}\ [\text{LIT}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x}\ [\text{VAR}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x!}\ [\text{REUSE}_C]$$

$$\frac{}{(\{x\}, R, \emptyset) \vdash_{check} x@r}\ [\text{COPY1}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x@r}\ [\text{COPY2}_C] \qquad \frac{}{(\emptyset, R, \{x\}) \vdash_{check} x@r}\ [\text{COPY3}_C]$$

$$\frac{}{(\emptyset, R, \emptyset) \vdash_{check} C\ \overline{a_i}^n\ @r}\ [\text{CONS}_C] \qquad \frac{f\overline{a_i}^n@r \vdash_{inf} (D, R, S, N) \quad \forall a_i \in D_p\,.\,(\#j : 1 \le j \le n : a_i = a_j) = 1}{(D_p, R_p, S_p) \vdash_{check} f\ \overline{a_i}^n@r}\ [\text{APP}_C]$$

$$\frac{\begin{array}{c} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \quad (D_p \cap N_1, R_p - D_1, S_p \cap N_1) \vdash_{check} e_1 \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad (D_p \cap N_2, (R_p \cup R'_p) - D_2, S_p \cap N_2) \vdash_{check} e_2 \\ R'_p = \{y \in ((D_p \cap N_1) \cup D_1) \cap sharerec(z, e_2) \mid z \in D_p \cap N_2\} - (N_2 \cup \{x_1\}) \\ R_p \cap (S_1 \cup S_2) = \emptyset \qquad x_1 \notin S_2 \cup N_2 \ \vee\ \forall z \in D_p \cap N_2\,.\,x_1 \notin sharerec(z, e_2) \\ L = FV(e_2) \qquad C = \begin{cases} shareall(x_1, e_2) & \text{si } x_1 \in S_2 \\ shareall(x_1, e_2) - sharerec(x_1, e_2) & \text{si } x_1 \in D_2 \end{cases} \\ R''_p = \{sharerec(z, e_1) \mid z \in D_p \cap N_1\} \\ def((D_p \cap N_1) \cup (R_p - D_2) \cup R''_p \rhd^L_C (D_p \cap N_2) \cup ((R_p \cup R'_p) - D_2)) \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}\ [\text{LET}_C]$$

$$\frac{\begin{array}{c} \forall i \in \{1..n\}\,.\,e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad \forall i \in \{1..n\}\,.\,D_{p_i} = \emptyset \\ \forall i \in \{1..n\}\,.\,P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad \forall i \in \{1..n\}\,.\,R_{p_i} = \begin{cases} Rec_i & \text{si } x \in D_p \\ \emptyset & \text{e.o.c.} \end{cases} \\ \forall i \in \{1..n\}\,.\,Rec_i = \bigcup_{j=1}^{n_i} \{x_{ij} | j \in RecPos(C_i)\} \\ type(x) = \begin{cases} d & \text{si } x \in D_p \\ r & \text{si } x \in R_p \\ s & \text{si } x \in S_p \\ n & \text{e.o.c.} \end{cases} \qquad \forall i \in \{1..n\}\,.\,S_{p_i} = \begin{cases} P_i - Rec_i & \text{si } x \in D_p \\ P_i & \text{si } x \in S_p \\ P_i - (R_i \cup S_i) & \text{si } x \in R_p \\ \emptyset & \text{e.o.c.} \end{cases} \\ \forall i \in \{1..n\}\,.\,R'_{p_i} = \{x \in P_i \cap sharerec(z, e_i) \mid z \in (D_p \cup D_{p_i}) \cap N_i\} \\ \forall i \in \{1..n\}\,.\,R''_{p_i} = \{y \in ((D_p \cap N) \cup D) \cap sharerec(z, e_i) \mid z \in (D_p \cup D_{p_i}) \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\,R'_{p_i} \cap (S_i \cup S_{p_i}) = \emptyset \ \wedge\ R_p \cap S_i = \emptyset \\ \forall i \in \{1..n\}\,.\,((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \\ x \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\}\,.\,def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case}\ x\ \textbf{of}\ \overline{C_i\overline{x_{ij}}^{n_i} \to e_i}^n}\ [\text{CASE}_C]$$

$$\frac{\begin{array}{c} \forall i \in \{1..n\}\,.\,e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\}\,.\,R'_{p_i} = \{x \in P_i \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} \\ \forall i \in \{1..n\}\,.\,R''_{p_i} = \{y \in ((D_p \cap N) \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\,(R'_{p_i} \cup R_p) \cap S_i = \emptyset \\ \forall i \in \{1..n\}\,.\,(D_p \cap N_i, (R_p \cup R'_{p_i} \cup R''_{p_i}) - D_i, S_p \cap N_i) \vdash_{check} e_i \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\overline{x_{ij}}^{n_i} \to e_i}^n}\ [\text{CASE!}_C]$$

Figura 5.8: Reglas de comprobación *top-down*

Si (D, R, S, N) son los conjuntos obtenidos por el algoritmo en su última iteración, es necesario realizar las siguientes comprobaciones finales tras alcanzar el punto fijo:

1. El conjunto R no puede contener ningún parámetro formal de la función, ya que este parámetro quedaría en peligro. Dado que siempre se cumple $R \subseteq scope(e)$ (ver Sección 6.2) y $scope(e) = \{x_1, \ldots, x_n\}$, basta con comprobar que se cumple $R = \emptyset$.

2. Como ya fue señalado en el Capítulo 2, el programador puede especificar un tipo anotado junto con la definición de la función. En dicho tipo es posible indicar (mediante la notación !) qué parámetros van a ser destruídos y, por tanto, tienen tipo condenado. En la inferencia Hindley-Milner ya se apuntó que el tipo anotado debía ser más particular y más restrictivo que el tipo inferido y se proporcionaron los mecanismos para realizar esta comprobación. En la inferencia SAFE es necesario comprobar la compatibilidad de la marca especificada por el usuario con la marca inferida por el algoritmo:

   - Si los parámetros marcados por el usuario como condenados coinciden con los inferidos por el algoritmo no es necesaria ninguna comprobación.
   - Si un parámetro fue inferido como condenado pero fue anotado como seguro por el programador debe considerarse el tipo anotado como incorrecto. El algoritmo devolverá error en este caso.
   - Si un parámetro x_j se infiere con marca N pero fue anotado como condenado por el programador, se le asigna a dicho parámetro tipo condenado y se realiza un $\vdash_{check}$ sobre la expresión de la función para comprobar que x_j puede ser condenado en la definición:

     $$(\{x_j\}, \emptyset, \emptyset) \vdash_{check} e.$$

     Si existen varios parámetros en esta misma situación puede realizarse el $\vdash_{check}$ simultáneamente sobre todos ellos.

3. Si tras el cálculo del punto fijo y finalizado el paso anterior, algún parámetro queda con marca N, se le fuerza a tener tipo seguro asignándole una marca S, que mediante las reglas de $\vdash_{check}$ será propagada por el resto del árbol abstracto: $(\emptyset, \emptyset, N) \vdash_{check} e$. Este check puede combinarse con el comentado en el punto anterior.

5.3.4. Coste del algoritmo

Si m es el número de parámetros de una función y n es el número de nodos del árbol abstracto que representa la definición de la misma, el algoritmo puede efectuar en el caso peor m iteraciones (recorridos *bottom-up*) hasta alcanzar el punto fijo. Por otro lado, un recorrido *bottom-up* puede verse interrumpido en cada nodo para hacer

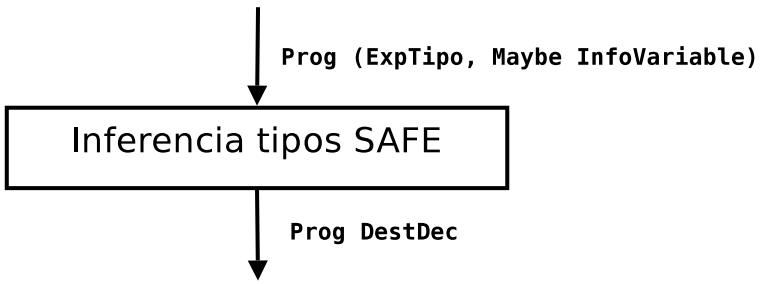

Figura 5.9: Entrada y salida de la inferencia de tipos SAFE

un $\vdash_{check}$. Por tanto, en el caso peor un nodo puede ser visitado n^2 veces durante una iteración. Dado que durante el algoritmo se realizan operaciones de unión entre conjuntos de variables y el número de variables de un programa en el caso peor es lineal con respecto al tamaño del árbol abstracto, se tiene que una sola iteración tiene coste de $\mathcal{O}(n^3)$ en el caso peor. Esto conlleva, también en el caso peor, un coste en tiempo de $\mathcal{O}(mn^3)$ para analizar una definición. No obstante, creemos que el caso medio está cerca de $\mathcal{O}(n^2)$, que corresponde a un único recorrido ascendente del árbol y una única iteración en el cálculo del punto fijo.

5.4. Detalles de implementación

Una vez presentado el algoritmo de inferencia, pasamos a describir su implementación en Haskell. Recordemos del esquema de la Figura 2.5 que la inferencia de tipos SAFE se realizaba tras la fase del análisis de compartición. Por ello la fase de inferencia de tipos SAFE recibe como entrada el árbol abstracto resultante de dicho análisis de compartición (Figura 5.9). Tras el algoritmo de inferencia el árbol abstracto quedará decorado con un valor de tipo `DestDec` que contiene, entre otra información, los cuatro conjuntos (D, R, S, N) inferidos para cada elemento del árbol.

5.4.1. Entorno de parámetros condenados

La inferencia se realiza de forma modular. El resultado de analizar cada función es, además de los conjuntos (D, R, S, N) con los que el árbol abstracto queda decorado, la signatura de dicha función. Ya se ha comentado que dicha signatura se compone de cuatro conjuntos de enteros (I_D, I_R, I_S, I_N) que indican la marca de cada parámetro. Dado que el conjunto I_R siempre ha de ser vacío, no se incluirá su información en la signatura. El tipo de datos `DestSig` queda definido, por tanto, como tres listas ordenadas de enteros que corresponden a los conjuntos I_D, I_S e I_N, respectivamente:

```
type DestPos = [Int]
type DestSig = (DestPos, DestPos, DestPos)
```

La signatura de todas las funciones se almacena en un entorno, definido como una tabla (`Data.Map`) en la que los nombres de función quedan asociados con signaturas:

```
type DestEnv = M.Map String DestSig
```

El análisis de una función recibe como entrada, además del árbol abtracto correspondiente, el entorno con la signatura de todas las funciones analizadas hasta el momento y produce como salida el mismo entorno ampliado con la nueva signatura calculada.

5.4.2. Definiciones auxiliares

Un vistazo rápido a las reglas de las Figuras 5.4 y 5.8 muestra la necesidad de definir funciones para las definiciones auxiliares contenidas en la mismas. Entre ellas se encuentran los operadores $\triangleright$ y $\sqcup$, las funciones *sharerec* y *shareall* y los predicados *inh* e *inh*!.

La implementación de los operadores $\triangleright$ y $\sqcup$ resulta bastante sencilla a partir de sus respectivas definiciones. Para el operador $\triangleright$ se define una función `disjointUnion`:

```
disjointUnion :: Variables -> Variables -> Variables -> Variables -> Bool
```

La llamada a `disjointUnion D1 D2 C L` equivale a $def(\texttt{D1} \triangleright^{\texttt{L}}_{\texttt{C}} \texttt{D2})$. Si el operador está definido se devuelve el valor `True`. En otro caso el algoritmo se detiene, devolviendo error. Con respecto a la operación de $\sqcup$, ésta queda implementada como una función `sqUnion`:

```
sqUnion :: [(Variables, Variables, Variables,
             Variables, Variables, Variables)]
        -> (Variables, Variables, Variables, Variables)
```

Esta función recibe como parámetro una lista con los $(D_i, R_i, S_i, N_i, P_i, Rec_i)$ de cada alternativa de un **case**(!) y devuelve los conjuntos (D, R, S, N) resultantes de reunir la información de todas las alternativas. Si hubiese incompatibilidad entre distintas alternativas la función devolvería error. En realidad la definición de $\sqcup$ no incluye las variables Rec_i, pero sí son pasadas como parámetro en la implementación de `sqUnion`, aunque luego no se utilicen. El motivo de esto es evitar la creación de una tupla intermedia de cinco elementos.

La información correspondiente a las funciones *shareall* y *sharerec* puede encontrarse en la decoración del árbol abstracto de entrada, que es de tipo `SharingDec`.

```
type SharingDec = (ExpTipo, Maybe InfoVariable)
```

En la Sección 4.5.5 se comentó que en la segunda componente de un par `SharingDec` podía almacenarse información de compartición relacionada con el elemento que quedaba decorado. Sólo determinados nodos del árbol abstracto tenían este tipo de decoración; el resto quedaba decorado con el valor `Nothing`. Por tanto, el resultado de

las funciones *sharerec* y *shareall* está contenido en la decoración del árbol abstracto. Tan sólo es necesario utilizar los selectores `ShRDir` y `ShDir` vistos en la Sección 4.5.1.

Con respecto a las funciones *inh* e *inh*!, éstas quedan implementadas mediante las funciones `definedInh` y `definedInhDest`, respectivamente.

```
definedInh :: VarTypeDest
           -> (Variables, Variables, Variables,
               Variables, Variables, Variables)
           -> Bool
definedInhDest
          :: (Variables, Variables, Variables,
              Variables, Variables, Variables)
           -> Bool
```

Ambas reciben como parámetro una tupla $(D_i, R_i, S_i, N_i, P_i, Rec_i)$, aunque el conjunto N_i no se utiliza realmente. La primera función recibe además el tipo SAFE del discriminante del **case** correspondiente.

```
data VarTypeDest = DType  -- Condenado
                 | RType  -- En peligro
                 | SType  -- Seguro
                 | NType  -- No determinado
```

En las reglas de $\vdash_{inf}$ también pueden encontrarse llamadas a la función *RecPos*, que devuelve las posiciones recursivas de un constructor de datos. En la implementación de esta función se utilizará un enfoque similar al del análisis de compartición (Sección 4.5.2), consistente en la creación previa de una tabla de posiciones recursivas, de tipo `TablaRecPos`.

5.4.3. Decoración del árbol abstracto resultante

Ya se ha visto que durante la inferencia de tipos SAFE se decoraba cada elemento del árbol abstracto de sintaxis con los cuatro conjuntos (D, R, S, N) que eran inferidos, con lo que el tipo de la decoración del árbol abstracto obtenido por el algoritmo podría definirse de la siguiente forma:

```
data DestDec = (Variables, Variables, Variables, Variables)
```

No obstante, en ocasiones resulta recomendable añadir cierta información adicional a esta decoración por motivos de eficiencia. Por ejemplo, en la regla de inferencia [LET$_I$] puede observarse una llamada a la función FV, que calcula las variables libres de una expresión. Esta operación resulta relativamente costosa, ya que requiere un recorrido de la expresión cuyas variables libres quieren recolectarse. Resultaría adecuado almacenar en algún lugar el resultado de una llamada a FV para no tener que calcularlo otra vez cuando se necesite (por ejemplo, en un posible check posterior).

En resumen, para evitar recalcular conjuntos y funciones innecesariamente se opta por incluir una **decoración dependiente de la expresión** a cada elemento del árbol abstracto. La decoración resultante del mismo queda compuesta por:

- Una tupla con los conjuntos (D, R, S, N) resultantes de la inferencia. La unión de estos conjuntos es superconjunto de las variables libres de la expresión decorada.
- Una decoración dependiente de la expresión, `ExpDepDec`:
  - En nodos **let** del árbol abstracto se almacenarán, por un lado, las variables libres de la expresión principal del mismo (L en la regla [LET$_I$]) y, por otro lado, el conjunto C de la regla [LET$_I$].
  - En los nodos **case** se almacenará una lista con los conjuntos (P_i, Rec_i) de cada alternativa, que podrán ser utilizados en un $\vdash_{check}$ posterior.
  - En el resto de nodos se colocará una decoración dependiente de la expresión vacía (`EmptyEDD`).

El tipo final de la decoración queda de la siguiente forma:

```
data DestDec = DD (Variables, Variables, Variables, Variables) ExpDepDec

data ExpDepDec = LetEDD Variables Variables
               | CaseEDD [(Variables, Variables)]
               | EmptyEDD
```

Hay ciertos elementos del árbol abstracto sobre los que no se realiza inferencia. Ejemplo de ello son las variables de los patrones y los argumentos de una llamada a una función. A este tipo de elementos se les asociará una decoración vacía, devuelta por `destDecDummy`.

```
destDecDummy :: DestDec
destDecDummy = DD (S.empty, S.empty, S.empty, S.empty) EmptyEDD
```

5.4.4. Reglas de inferencia y check

La función que implementa las reglas $\vdash_{inf}$ y el recorrido *bottom-up* del árbol abstracto se llama `destInferenceExp` y su signatura se muestra a continuación:

```
destInferenceExp :: TablaRecPos -> DestEnv -> Exp SharingDec -> Exp DestDec
```

Esta función recibe como parámetros la tabla de posiciones recursivas de constructoras (para la función *RecPos*), el entorno de parámetros condenados y la expresión sobre la que se realiza el análisis. Devuelve como resultado la expresión decorada tal como se indicó en la Sección 5.4.3.

La implementación de esta función se realiza por distinción de casos según la estructura de la expresión e imita la especificación de las reglas de la Figura 5.4. Puede encontrarse la implementación correspondiente a la regla [CASE$_I$] en la Figura 5.10. En primer lugar se realiza la inferencia para cada alternativa, obteniendo por un lado las alternativas procesadas `alts'` y los conjuntos $(D_i, R_i, S_i, N_i, P_i, Rec_i)$ correspondientes cada una en la variable `sets`. Obtenemos la lista de expresiones pertenecientes a las alternativas, `exps`, se realiza el cálculo de los R'_i y S'_i (`riandsi'`) y con estos dos elementos se procede a realizar el $\vdash_{check}$ mediante la función `checkExp`. La expresión con

$$\frac{\begin{array}{l}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad (D, R, S, N) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \quad \forall i \in \{1..n\} \,.\, def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \\ type(x) = \begin{cases} d & \text{si } x \in D \\ r & \text{si } x \in R \\ s & \text{si } x \in S \\ n & \text{e. o. c.} \end{cases} \quad N' = \begin{cases} N & \text{si } x \in D \cup R \cup S \\ N \cup \{x\} & \text{si } x \notin D \cup R \cup S \end{cases} \\ \forall i \in \{1..n\} \,.\, ((D \cup D_i') \cap N_i, (R \cup R_i') \cap N_i, (S \cup S_i') \cap N_i) \vdash_{check} e_i \\ \textbf{where } D_i' = \emptyset \quad R_i' = \begin{cases} \emptyset & \text{si } type(x) = s \\ \emptyset & \text{si } type(x) = r \\ Rec_i & \text{si } type(x) = d \end{cases} \quad S_i' = \begin{cases} P_i & \text{si } type(x) = s \\ \emptyset & \text{si } type(x) = r \\ P_i - Rec_i & \text{si } type(x) = d \end{cases} \end{array}}{\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \vdash_{inf} (D, R, S, N')} \text{ [CASE}_I\text{]}$$

```
destInferenceExp trp env exp@(CaseE disc@(VarE x _) alts _)
      | defined = result
   where (alts', sets) = unzip $ map (destInferenceAltCase trp env x) alts
         sqUn@(d,r,s,n) = sqUnion sets
         n' | S.member x d || S.member x r || S.member x s = n
            | otherwise = S.insert x n

         -- Comprobación del inh
         typeX = getTypeDest x sqUn
         defined = all (definedInh typeX) sets

         -- Comprobación del check
         (pats, exps) = unzip alts'
         riandsi' = map (g typeX) sets
         exps' = zipWith f riandsi' exps
         alts'' = zip pats exps'

         -- f se encarga de la llamada a check para cada alternativa
         f (ri',si') ex = checkExp (S.intersection d ni,
                                    S.intersection (S.union r ri') ni,
                                    S.intersection (S.union s si') ni) ex
              where (DD (_,_,_,ni) _) = decExp ex

         -- g calcula los ri', si' de la regla de CASE
         g DType (di,ri,si,ni,pi,reci) = (reci, pi S.\\ reci)
         g RType (di,ri,si,ni,pi,reci) = (pi, S.empty)
         g SType (di,ri,si,ni,pi,reci) = (S.empty, pi)
         g NType _ = (S.empty, S.empty)

         result = CaseE (VarE x destDecDummy) alts'' (DD (d,r,s,n')
                   (CaseEDD (map (\(_,_,_,_,pi,reci)->(pi, reci)) sets)))
```

Figura 5.10: Regla [CASE$_I$] y su implementación

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\} \,.\, (D_p \cap N_i, R_p \cap N_i, S_p \cap N_i) \vdash_{check} e_i\end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case!}\ x\ \textbf{of}\ \overline{C_i \overline{x_{ij}}^{n_i} \to e_i}^{n}} \text{ [CASE!}_C\text{]}$$

```
checkExp new@(dp,rp,sp) (CaseDE x patExps (DD old edd))=
                (CaseDE x (zip pats exps') (DD (changeColour new old) edd))
   where (pats,exps) = unzip patExps
         newVars = map (f . decExp) exps
         f (DD (_,_,_,ni) _) = (S.intersection dp ni,
                                S.intersection rp ni,
                                S.intersection sp ni)
         exps' = zipWith checkExp newVars exps
```

Figura 5.11: Regla [CASE!$_C$] y su implementación

la decoración resultante se encuentra en la definición de `result`, que será devuelta si la variable `defined` tiene valor `True`. Para ello es necesario comprobar que la relación *inh*! está definida mediante la función `definedInh` vista anteriormente.

Dentro de la definición de la función `destInferenceExp` aparecen llamadas a la función `checkExp` que implementa las reglas de $\vdash_{check}$. Esta función tiene el siguiente tipo:

```
checkExp :: (Variables, Variables, Variables) -> Exp DestDec -> Exp DestDec
```

El primer parámetro representa los conjuntos (D_p, R_p, S_p) con variables con las que se realizará la comprobación. El segundo parámetro es la expresión a analizar. Si el check tiene éxito se devolverá la expresión con la decoración actualizada: las variables de D_p (resp. R_p, S_p) que pertenecían al conjunto N pasan a pertenecer a D (resp. R, S). En la Figura 5.11 puede encontrarse la implementación de la regla [CASE!$_C$].

5.4.5. Cálculo del punto fijo y comprobaciones finales

La inferencia de tipos seguros para la definición de una función requería, además de los recorridos del árbol abstracto dirigidos por las reglas $\vdash_{inf}$ y $\vdash_{check}$, una serie de comprobaciones adicionales explicadas en la Sección 5.3.3. Dichas comprobaciones son realizadas por la función `destInferenceDef'`, que realiza repetidamente llamadas a `destInferenceExp` hasta alcanzar el punto fijo.

```
destInferenceDef' :: TablaRecPos -> DestEnv -> (Variables, Variables)
                  -> Def SharingDec -> (DestEnv, Def DestDec, DestSig)
```

Esta función recibe, además de la definición a procesar, la tabla de posiciones recursivas, el entorno de parámetros condenados y un par (D_u, S_u) que indican los nombres de los parámetros que el usuario ha anotado como condenados y seguros. En la Figura 5.12 podemos encontrar su implementación.

```
destInferenceDef' trp env sigU@(dUser,sUser) def@(ts,(nom, pbs, rs),
                                                          Simple e [])
   = if (sigN == env M.! nom)
      then (M.insert nom sigN' env,
                              (ts, (nom, pbs', rs), Simple e'' []), sigN')
      else (destInferenceDef' trp (M.insert nom sigN env) sigU def)
 where patsList = map fst pbs
       e' = destInferenceExp trp env e
       DD (d,r,s,n) _ = decExp e'
       rEmpty = S.null r || error ("R not empty: " ++ show r)
       dUserAndS = dUser `S.intersection` s
       dUserOK = S.null dUserAndS
                 || error ("Parameters must not be condemned: "
                           ++ show dUserAndS)
       sUserAndD = sUser `S.intersection` d
       sUserOK = S.null sUserAndD
                 || error ("Parameters must be condemned: "
                           ++ show sUserAndD)
       sigN@(dpos,spos,npos) | rEmpty && dUserOK && sUserOK
              = (extract patsList d, extract patsList s, extract patsList n)
       pbs' = [(setDecPat destDecDummy p,b) | (p,b) <- pbs]
       e'' = checkExp (n `S.intersection` dUser, S.empty, n S.\\ dUser) e'
       DD (d',_,s',_) _ = decExp e''
       sigN' = (extract patsList d', extract patsList s', [])
```

Figura 5.12: Implementación de la función `destInferenceDef'`

Tras realizar la inferencia de la expresión `e` se obtiene la tupla `(d,r,s,n)` como resultado. Tras comprobar que el conjunto de variables en peligro es vacío (`rEmpty`) y que el tipo anotado por el usuario (si lo hay) es compatible con el inferido (`dUserOK` y `sUserOK`) se extrae la signatura de la función `sigN`. Si la signatura obtenida es distinta a la que ya se encontraba en el entorno es necesaria otra iteración. En caso contrario se realiza el último $\vdash_{check}$ con las variables indicadas por el usuario como destruibles y las variables que fueron inferidas como N. Con ello se obtiene la expresión final `e''` de la que se extrae la signatura final `sigN'`.

La llamada inicial a `destInferenceDef'` debe hacerse con la signatura en la que todos los parámetros tienen marca N. Esto es realizado por la función `destInferenceDef`, que devuelve la definición de la función y el entorno actualizados.

```
destInferenceDef :: TablaRecPos ->  DestEnv -> Def SharingDec
                 -> (DestEnv, Def DestDec)
```

Por último, la función `destInferenceProg` es la encargada de aplicar la función anterior a cada definición del programa:

```
destInferenceProg :: DestEnv -> Prog SharingDec
                  -> (DestEnv, Prog DestDec)
destInferenceProg env (decsData, defs, exp) = (env',
                       (decsData, defs', destInferenceExp trp env' exp))
```

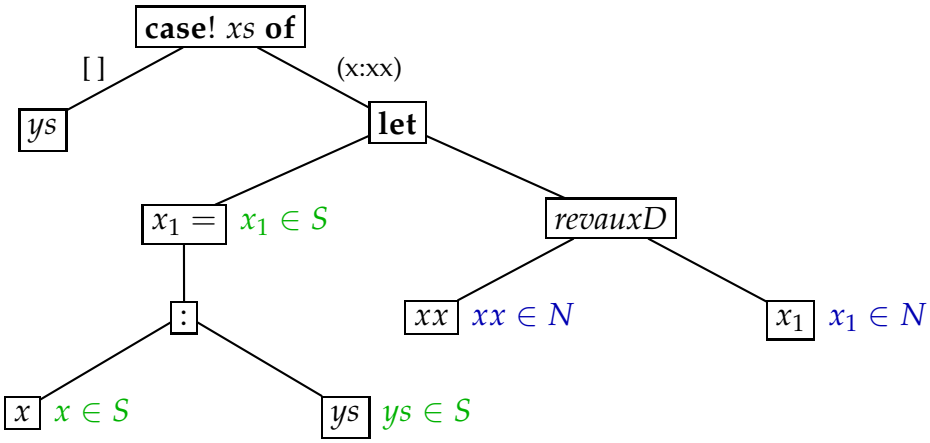

Figura 5.13: Análisis de la alternativa $(x : xx)$ en *revAuxD*.

```
where trp = construirTablaRecPos decsData
      (env', defs') = L.mapAccumL (destInferenceDef trp) env defs
```

5.5. Ejemplo

Se mostrará el funcionamiento del algoritmo de inferencia mediante su aplicación a la siguiente función *revauxD*, versión con parámetro acumulador de la función que invierte una lista, destruyendo la lista de entrada:

$$
\begin{array}{l}
\mathit{revauxD}\ xs\ ys\ @r = \\
\quad \textbf{case}!\ xs\ \textbf{of} \\
\quad\quad [\,] \rightarrow ys \\
\quad\quad (x : xx) \rightarrow \textbf{let}\ x_1 = (x : ys)@r\ \textbf{in}\ \mathit{revauxD}\ xx\ x_1\ @r
\end{array}
$$

Primera iteración

Inicialmente suponemos que *revauxD* tiene la siguiente signatura: $(\emptyset, \emptyset, \emptyset, \{1, 2\})$. Es decir, sus dos parámetros tienen marca N. Por tanto, las dos variables xx y x_1 pasadas como parámetro en la llamada recursiva adquieren también marca N. Por otro lado, examinando la expresión auxiliar del **let** se obtiene que las variables x e ys adquieren marca S, ya que son utilizadas para construir una estructura de datos (regla [CONS$_I$] en Figura 5.4). La variable ligada x_1 también adquiere tipo S, ya que no se utiliza destructivamente en la expresión principal del **let**. La asignación en este punto del algoritmo se refleja en la Figura 5.13.

Reuniendo los resultados de la expresión auxiliar y principal del **let** se obtiene que la variable x_1 tiene marca S en la definición auxiliar y marca N en la expresión principal. Es necesario propagar la marca S a lo largo de la llamada recursiva mediante las reglas $\vdash_{check}$. Por tanto, es necesario derivar:

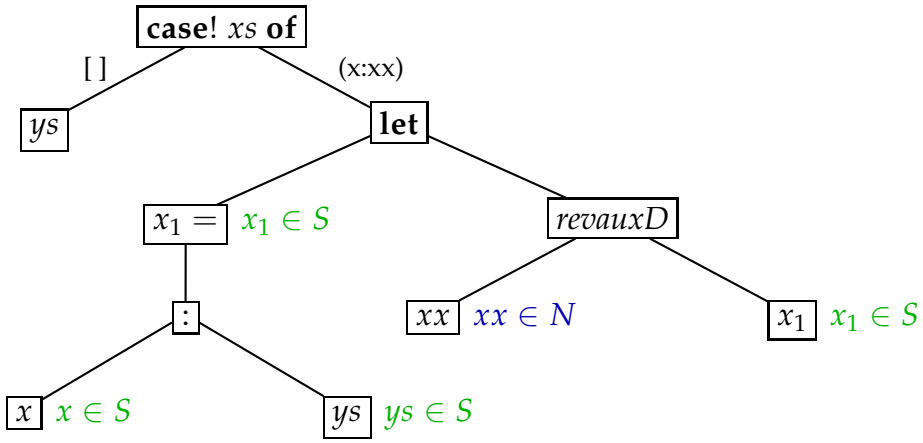

Figura 5.14: Análisis de la alternativa $(x : xx)$ en *revAuxD* tras la comprobación.

$$(\emptyset, \emptyset, \{x_1\}) \vdash_{check} revauxD\ xx\ x_1\ @r$$

La comprobación tiene éxito y se actualiza la decoración correspondiente, obteniendo la situación de la Figura 5.14.

La inferencia de la alternativa del **case**! guardadada mediante $[\,]$ es más sencilla. La expresión consiste en una única variable que obtiene marca S. Si llamamos $e_{[\,]}$ y $e_{(x:xx)}$ a las expresiones contenidas en las respectivas alternativas del **case**! externo, el algoritmo ha obtenido hasta ahora:

$$\begin{array}{rll} e_{[\,]} & \vdash_{inf} & (\emptyset, \emptyset, \{ys\}, \emptyset) \\ e_{(x:xx)} & \vdash_{inf} & (\emptyset, \emptyset, \{x, ys\}, \{xx\}) \end{array}$$

Para poder aplicar la regla de inferencia [CASE!$_I$] debe estar definido el operador $\sqcup$, que reune todas las alternativas. La única variable que podría resultar conflictiva es ys, ya que aparece en las dos alternativas. Sin embargo, dado que en ambas obtiene marca S la información de las alternativas es compatible, la operación $\sqcup$ está definida y tiene como resultado $(\emptyset, \emptyset, \{ys\}, \emptyset)$, en el que las variables x y xx han sido eliminadas, al estar ligadas en la segunda alternativa. Por otro lado, la variable xs adquiere marca D, ya que es el discriminante de un **case** destructivo. La situación actual se muestra en la Figura 5.15.

Durante el análisis del **case**! externo también se asigna la marca heredada D a la variable xx, ya que es hijo recursivo de xs. Por tanto es necesario un $\vdash_{check}$ sobre la segunda alternativa:

$$(\{xx\}, \emptyset, \emptyset) \vdash_{check} e_{(x:xx)}$$

La comprobación actualiza la decoración del árbol abstracto (Figura 5.16). Si e es la expresión total del lado derecho de la definición, se ha obtenido:

$$e \vdash_{inf} (\{xs\}, \emptyset, \{ys\}, \emptyset)$$

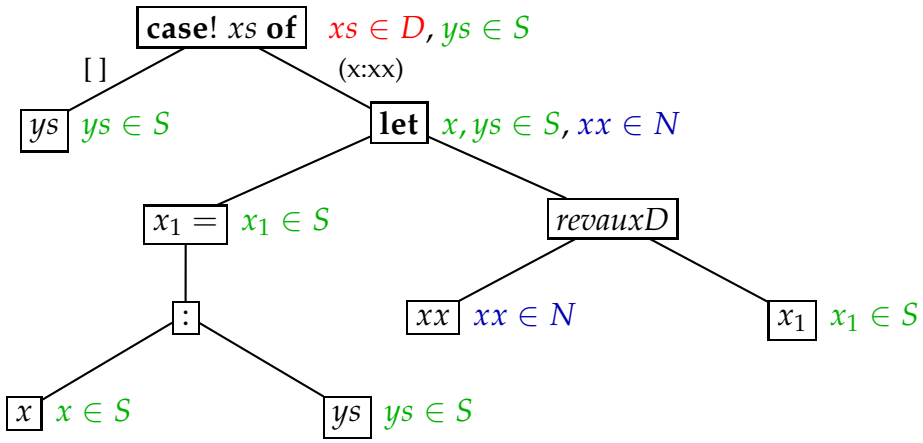

Figura 5.15: Análisis del **case**! externo.

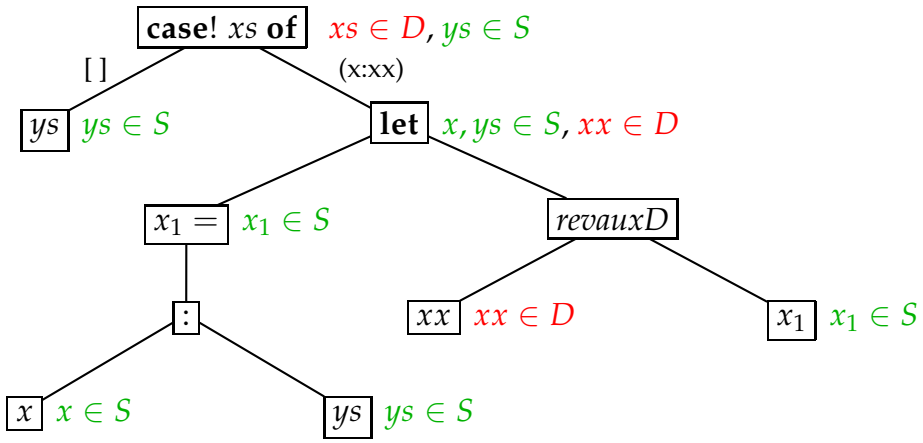

Figura 5.16: Análisis del **case**! externo tras la comprobacion.

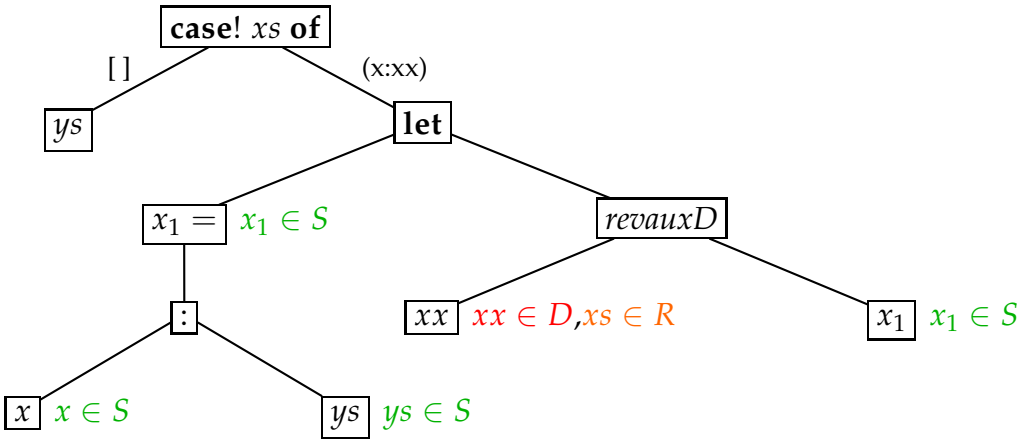

Figura 5.17: Análisis de la alternativa $(x : xx)$ en la segunda iteración.

A partir de este resultado se extrae la signatura $(\{1\}, \emptyset, \{2\}, \emptyset)$. Dado que difiere de la signatura inicial, no se ha alcanzado el punto fijo y, por tanto, es necesario repetir el proceso.

Segunda iteración

Se procede a analizar la llamada recursiva a *revauxD*. A partir de la signatura de esta función se conoce que xx y x_1 están marcadas como D y S respectivamente. Además xs pasa a estar en peligro, ya que al ser padre de xx, comparte un hijo recursivo de éste. Por otro lado, el análisis de la expresión auxiliar del **let** es idéntico al de la iteración anterior (Figura 5.17). Aplicando la regla [LET$_I$] se obtiene un conjunto C, al cual xs pertenece. Esto podría plantear un problema, ya que xs es inferida como R en la expresión principal del **let**. No obstante, no hay ninguna aparición libre de xs en dicha expresión principal y, por tanto, el operador $\rhd_C^L$ está bien definido en este caso.

En la Figura 5.18 puede encontrarse la situación final tras la segunda iteración. Tras analizar la expresión total e, se ha obtenido el siguiente resultado:

$$e \vdash_{inf} (\{xs\}, \emptyset, \{ys\}, \emptyset)$$

A partir de estos cuatro conjuntos se extrae la misma signatura que en la primera iteración y, por tanto, se ha alcanzado el punto fijo. El tipo final de la función *revauxD* es:

$$revauxD :: \forall a, \rho_1, \rho_2.[a]!@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2$$

Es decir, sólo la lista pasada como primer parámetro quedará destruida tras la llamada a la función.

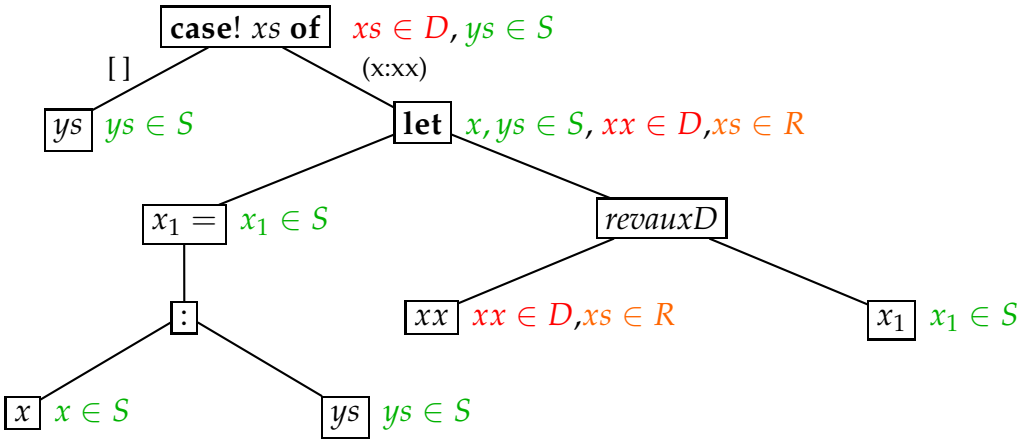

Figura 5.18: Situación final tras la segunda iteración.

5.6. Casos de estudio

Con el fin de estudiar el comportamiento del algoritmo de inferencia explicado en este capítulo, se codificaron varios ejemplos en lenguaje *Full-Safe* y fueron analizados por el compilador. Las funciones creadas implementan los algoritmos básicos que trabajan sobre listas (como por ejemplo ordenación) y sobre árboles de búsqueda binarios (inserciones, eliminaciones, recorridos, etc.) En el caso de definiciones recursivas sencillas, en los que no se realiza llamada a ninguna otra función, el algoritmo obtuvo resultados satisfactorios. Es decir, las definiciones fueron admitidas por el algoritmo y el tipo inferido se corresponde con el que cabría esperar considerando el comportamiento de la función.

EJEMPLO 17. *Se muestran a continuación las versiones destructivas de algunas de estas funciones sencillas:*

```
-- Longitud de una lista
lengthD [ ]! = 0
lengthD (x : xs)! = 1 + lengthD xs

-- Cabeza de una lista
headD (x : xs)! = x

-- Resto de una lista
tailD (x : xs)! = xs!
```

```
-- Último elemento de una lista
lastD (x : [])! = x
lastD (x : xs)! = lastD xs
-- Concatenar listas
concatD []! xs@r = xs
concatD (x : xs)! ys@r = (x : (concatD xs ys @r))@r
```

Los siguientes tipos fueron inferidos por el compilador:

$$
\begin{array}{lcl}
lengthD & :: & [a]!@\rho \to Int \\
headD & :: & [a]!@\rho \to a \\
tailD & :: & [a]!@\rho \to [a]@\rho \\
lastD & :: & [a]!@\rho \to a \\
concatD & :: & [a]!@\rho_1 \to [a]@\rho_1 \to \rho_2 \to [a]@\rho_2
\end{array}
$$

Como puede verse, el primer parámetro aparece con tipo condenado en todos los casos, ya que se destruye la lista de entrada.

Con respecto a las funciones que implementan la ordenación de listas y que requieren el uso de otras definiciones auxiliares, los resultados obtenidos no son igual de satisfactorios: el algoritmo rechaza algunos programas, aún cuando estos realizan las destrucciones de forma segura.

EJEMPLO 18 (Mergesort)**.** *Retomemos el ejemplo introducido en la Sección 2.3 que implementa el algoritmo de ordenación por mezclas. Existen dos funciones auxiliares: splitD, que divide la lista de entrada en dos sublistas, y mergeD, que se encarga de reunir dos listas ordenadas en una sola lista. En ambos casos se destruye la lista de entrada. Estas funciones auxiliares son aceptadas por el algoritmo e inferidas con los tipos que se indican a continuación:*

$$
\begin{array}{lcl}
splitD & :: & Int \to [a]!@\rho \to ([a]@\rho, [a]@\rho)@\rho \\
mergeD & :: & [Int]!@\rho \to [Int]!@\rho \to \rho \to [Int]@\rho
\end{array}
$$

Puede observarse de que mergeD no es polimórfica con respecto al tipo de las listas de entrada. Esto es debido a que en su definición se hace uso del operador $<$, que temporalmente está definido sobre enteros en el preludio de SAFE. En este momento el lenguaje no incorpora ningún sistema de clases al estilo de Haskell que permita tipos de la forma Ord $a \Rightarrow a$.

La función msortD, que hace uso de estas funciones auxiliares, no es admitida por el algoritmo. Esto tiene su origen en el análisis de compartición, que al ofrecer una aproximación por exceso *de las relaciones de compartición implicadas en la llamada a una función, puede devolver como resultado más comparticiones de las que realmente podrían llegar a ocurrir durante la ejecución de la misma. En el caso de ordenación por mezclas partimos del siguiente código*

Full-Safe para la función msortD:

```
msortD xs @r
      | n ≤ 1  =  xs!
      | n > 1  =  mergeD (msortD xs_1 @r) (msortD xs_2 @r) @r
          where
            n          =  length xs
            (xs_1, xs_2)  =  splitD (n/2) xs @r
```

A continuación se incluye un fragmento del código Core-Safe[2] generado:

```
msortD xs =
    let n = length xs in
    let p = (let n_2 = n/2 in splitD n_2 xs) in
    let xs_1 = case p of (x_1, x_2) → x_1 in
    let xs_2 = case p of (x'_1, x'_2) → x'_2 in
    let c = n ≤ 1 in
        case c of
            True → ...
            False →
                let c' = n > 1 in
                    case c' of
                        False → ...
                        True → let s_1 = msortD xs_1 in
                               let s_2 = msortD xs_2 in
                                                mergeD s_1 s_2
```

Durante el análisis de compartición se detecta lo siguiente:

1. *En la expresión (***let** $xs_1 = \ldots$*) se obtiene* $x_1 \lhd p$.
2. *Por tanto,* $xs_1 \lhd p$.
3. *De modo análogo, en la expresión (***let** $xs_2 = \ldots$*) se obtiene* $x'_2 \lhd p$.
4. *Sin embargo, en ese mismo* **case**, *la definición de* Sub_i *en la interpretación S indica que* x'_2 *puede ser padre de todos los hijos de p, en concreto de* xs_1. *Por tanto,* $xs_1 \lhd x'_2 \lhd p$.
5. *Esto hace que* $xs_1 \lhd xs_2$.
6. *Al tener* $xs_1 \lhd xs_2$, *se produce la relación* $xs_1 \triangle xs_2$ *(y también* $xs_2 \triangle xs_1$*).*
7. *Cuando se analiza la llamada (msortD* xs_1*), se infieren* $xs_1 \triangle\!\!\sim s_1$ *y* $xs_2 \triangle\!\!\sim s_1$. *Esto último es debido a que* xs_2 *es padre de* xs_1.

[2]Se omiten variables de región, por motivos de claridad

8. *Por tanto,* $xs_1 \triangle s_1$, $xs_2 \triangle s_1$. *De esto último se obtiene, por simetría,* $s_1 \triangle xs_2$.

9. *En la llamada a (msortD* xs_2*), tenemos que* $s_1 \triangle\!\!\sim s_2$ *(ya que* s_1 *comparte un hijo con* xs_2*).*

Cuando el algoritmo de inferencia de tipos seguros procesa la llamada a mergeD obtiene que $s_1 \in D$*, ya que está en una posición condenada, pero también* $s_1 \in R$*, al pertenecer a* $sharerec(s_2, mergeD\ s_1\ s_2)$*, donde* s_2 *también se encuentra en una posición condenada. Como la regla [*APP_I*] requiere que los conjuntos D y R sean disjuntos, el algoritmo de inferencia no admite la definición.*

El ejemplo anterior muestra cómo una imprecisión proveniente del análisis de compartición ha tenido como consecuencia el rechazo de un programa válido. Con la función que implementa el algoritmo de ordenación *Quicksort* ocurre un problema similar. La fuente de este tipo de imprecisiones procede en gran parte de la definición de la interpretación S del análisis de compartición para el caso de la aplicación de una función. Recordemos dicha definición:

$$\begin{array}{rcl} S\ [\![g\ \overline{a_i}^m @r]\!]\ SubR\ ShR\ Sub\ Sh\ \rho & = & (\{z \mid \exists j \in SubRPg.a_j \in SubR(z)\}, \\ & & \{\mathbf{z} \mid \exists \mathbf{j} \in \mathbf{Shg}\,.\,\mathbf{a_j} \in \mathbf{ShR(z)}\}, \\ & & \{z \mid \exists j \in SubPg.a_j \in Sub(z)\}, \\ & & \bigcup_j \{SubR(a_j) \mid j \in SubRg\}, \\ & & \bigcup_j \{Sh(a_j) \mid j \in ShRg\}, \\ & & \bigcup_j \{Sub(a_j) \mid j \in Subg\}, \\ & & \bigcup_j \{Sh(a_j) \mid j \in Shg\}) \end{array}$$

$$\textbf{where}\ (SubRPg, ShRPg, SubPg, SubRg, ShRg, Subg, Shg) = \rho(g)$$

El análisis de compartición mantiene una distinción entre hijos recursivos e hijos no recursivos. La información obtenida sobre hijos recursivos es más precisa que la obtenida por hijos no recursivos. Sin embargo, el conjunto resaltado en la definición anterior representa el conjunto de variables que tienen un hijo *recursivo* al cual el resultado de la aplicación apunta. La imprecisión que esto generaba ya se comentó en la Figura 4.3. En lineas generales se tiene que la información referente a hijos recursivos resulta "contaminada" por la información de hijos no recursivos, más general que la primera.

Capítulo 6

Corrección del algoritmo de inferencia

El algoritmo de inferencia presentado en capítulos anteriores se dividía en dos fases. Por un lado se hallaban los tipos Hindley-Milner de cada definición y cada expresión de un programa *Full-Safe* mediante el proceso de generación de ecuaciones de unificación entre tipos expuesto en el Capítulo 3. Posteriormente los tipos eran ampliados con información sobre el uso previsto de una determinada estructura de datos; de este modo se distinguía entre tipos seguros (s), en peligro (r) y condenados (d). El algoritmo presentado en el Capítulo 5 se encargaba de asignar una de estas tres marcas a las variables del programa. Si en un determinado momento de la inferencia no podía conocerse la marca de una variable se le asignaba la marca desconocida (n), que posteriormente era sustituida en un mphcheck posterior.

Por otra parte también se disponía en SAFE de un sistema de tipos (Sección 5.2) que garantizaba el uso correcto de memoria de un programa. Si un programa admite un tipado mediante las reglas de la Figura 5.3 durante la ejecución del mismo no se producirán accesos a estructuras de datos que hayan sido previamente liberadas.

El objetivo de este capítulo es demostrar la corrección del algoritmo de inferencia con respecto al sistema de tipos. No obstante, la demostración de corrección se limitará únicamente al carácter seguro, condenado o peligro de un determinado tipo, ignorándose el tipo Hindley-Milner correspondiente. Por tanto sólo será necesario centrarse en el algoritmo de inferencia presentado en el Capítulo 5. A continuación, y a modo de recordatorio, se incluye un resumen del funcionamiento del mismo. Dada una definición $f\ \overline{x_i}^n = e$:

1. Se realiza un recorrido ascendente del árbol abstracto dirigido por las reglas $\vdash_{inf}$ de la Figura 5.4. Con ello se obtienen cuatro conjuntos (D,R,S,N) que indican la marca de las variables en ámbito de e, mientras que cada subexpresión de e queda decorada con los cuatro conjuntos correspondientes. Este recorrido puede requerir comprobaciones de consistencia a medida que una variable con tipo n adquiere cualquiera de los otros tres tipos en un contexto superior. Para ello se realiza un recorrido en sentido descendente dirigido por las reglas $\vdash_{check}$, presentadas en la Figura 5.8. Este recorrido produce también un cambio en la decoración del nodo sobre el que se hace mphcheck.

2. Se comprueba que $R = \emptyset$, ya que ningún parámetro de entrada a la función ha de tener tipo r.

3. Si la función que se está analizando es recursiva es necesario repetir los pasos 1 y 2 hasta alcanzar el punto fijo en la signatura de f.

4. Una vez alcanzado el punto fijo se hace último $\vdash_{check}$, obligando que las variables inferidas con tipo n adquieran tipo s.

A continuación se introducirán algunas propiedades del sistema de tipos y del algoritmo de inferencia. Estas propiedades son necesarias para demostrar posteriormente la corrección de este último para funciones no recursivas. Finalmente se ampliará la demostración para incluir funciones recursivas.

6.1. Propiedades del sistema de tipos

El sistema de tipos SAFE está definido mediante un conjunto de reglas en las que se encuentran juicios de la forma $\Gamma^P \vdash e : \tau$, donde bajo un entorno Γ y una lista de parámetros P la expresión e tiene tipo τ. No obstante, como se verá a continuación, el sistema de tipos siempre asigna tipos seguros (s) a las expresiones, por lo que los juicios pueden expresarse directamente como $\Gamma^P \vdash e : s$.

LEMA 1. *Si* $\Gamma^P \vdash e : \tau$, *entonces* $safe?(\tau)$.

Demostración. Por inducción sobre la derivación de $\Gamma^P \vdash e : \tau$. En la aplicación de las reglas [VAR], [LIT], [COPY], [APP1] y [REUSE] se cumple trivialmente. En el resto de reglas puede obtenerse $safe?(\tau)$ directamente a partir de la hipótesis de inducción. □

En el Capítulo 5 se comentaba la existencia del siguiente invariante en el sistema de tipos: Si una variable x aparece con tipo d en el entorno, todas las variables que comparten un hijo recursivo de x deben constar en dicho entorno como no seguras, ya que las estructuras de datos a las que apuntan pueden quedar parcialmente destruídas. Esto se especifica en el sistema de tipos mediante la restricción de que las variables, literales y aplicaciones de símbolos de constructora o de función siempre tipan bajo un entorno mínimo. Si se quiere ampliar ese entorno sólo puede hacerse mediante las reglas [EXTS] y [EXTD], que mantienen este invariante.

LEMA 2. *Si* $\Gamma^P \vdash e : s$ *y* $\Gamma^P(x) = d$ *entonces:*

$$\forall y \in sharerec(x, e) - \{x\} \,.\, y \in dom(\Gamma^P) \wedge unsafe?(\Gamma^P(y))$$

Demostración. Por inducción sobre la derivación de $\Gamma^P \vdash e : s$.

En las reglas [LIT] y [VAR] se cumple trivialmente, al no existir variables con tipo d en el entorno. Si la última regla utilizada es [REUSE] sólo existirá una variable en el entorno con tipo d, pero todas las variables que pertenezcan a $sharerec(x, x!) - \{x\}$ constarán en el entorno Γ_R con tipo r. En el caso de la regla [COPY], si una variable x

con tipo d aparece en Γ_1^P, todas las variables que pertenezcan a $sharerec(x, x@r) - \{x\}$ también aparecerán inseguras en Γ_1^P, por definición del operador $\geq$.

En el caso de las regla [EXTS] las variables con tipo d se encuentran en Γ^P y por tanto, el invariante se cumple por hipótesis de inducción. Con la regla [EXTD] la variable x tiene tipo d, pero todas las variables contenidas en $sharerec(x, e) - \{x\}$ se incluyen en Γ_R con tipo r. Si en el dominio de Γ^P hay otra variable $z \neq x$, necesariamente tenemos $z \in dom(\Gamma^P)$, y en ese caso el invariante se cumple para z por hipótesis de inducción.

Para expresiones e de la forma **let** $x_1 = e_1$ **in** e_2 (reglas [LET1] y [LET2]) se tiene $\Gamma^P \equiv \Gamma_1^P \rhd_C^L \Gamma_2^P$. Sea $x \in dom(\Gamma^P)$ tal que $\Gamma^P(x) = d$. Distinguimos casos:

- $\Gamma^P(x) = \Gamma_1^P(x)$

  Por hipótesis de inducción $sharerec(x, e_1) - \{x\}$ aparece con tipo inseguro en Γ_1^P. Dado que $scope(e_1) = scope(e)$ se tiene que $sharerec(x, e_1) = sharerec(x, e)$. Además, si y tiene tipo inseguro en Γ_1^P también lo tendrá en $\Gamma_1^P \rhd_C^L \Gamma_2^P$, por definición del operador $\rhd_C^L$. Por tanto, $sharerec(x, e) - \{x\}$ aparece con tipo inseguro en Γ^P.

- $\Gamma^P(x) = \Gamma_2^P(x)$

  Por hipótesis de inducción $sharerec(x, e_2) - \{x\}$ aparece con tipo inseguro en Γ_2^P. En este caso tenemos $scope(e) = scope(e_2) - \{x_1\}$, por lo que:

  $$sharerec(x, e) \subseteq sharerec(x, e_2)$$

  Por tanto, $sharerec(x, e) - \{x\}$ también aparece con tipo inseguro en Γ_2^P y, por la definición de $\rhd_C^L$, en Γ^P.

En el caso de la aplicación de función (reglas [APP1] y [APP2]) se tiene $\Gamma^P \equiv \Gamma_R + \Gamma'^P$. Si $x \in dom(\Gamma^P)$ y $\Gamma^P(x) = d$, necesariamente tenemos $x \in dom(\Gamma'^P)$, ya que Γ_R sólo contiene variables de tipo r.

Dado que $x \in dom(\Gamma'^P)$, tenemos que $\Gamma'^P(x) = t_i$, para algún i. En tal caso se tiene

$$sharerec(x, e) - \{x\} \subseteq R$$

El conjunto $sharerec(x, e) - \{x\}$ aparece con tipo inseguro en Γ_R y, por tanto, en Γ^P.

En expresiones de la forma $C\ \overline{a_i}^n\ @r$ (regla [CONS]) el invariante se cumple trivialmente, ya que no hay ninguna variable con tipo d en Γ^P.

En la regla [CASE] el invariante se cumple por definición del operador $\geq$, que impone que si $\Gamma^P(x) = d$ entonces $sharerec(x, e) - \{x\}$ aparece con tipo inseguro en $\Gamma^P(x)$.

Con respecto a expresiones de la forma **case**! z **of** $\ldots$ (regla [CASE!]), sea $\Gamma^P = \Gamma_R \otimes \Gamma'^P + [z : T!@p]$. Tenemos que $x \in dom(\Gamma'^P)$, o bien $x = z$. En el primer caso el teorema se cumple por hipótesis de inducción. En el segundo caso se verifica por la inclusión del entorno Γ_R en Γ^P. en □

6.2. Propiedades del algoritmo de inferencia

Un aspecto importante del algoritmo de inferencia es el hecho de que a una variable libre en cada expresión se le asigna un único tipo d, r, s, ó n. Por tanto cada variable sólo puede pertenecer a un conjunto D, R, S ó N. Además, los conjuntos D, S y N de las reglas de $\vdash_{inf}$ siempre involucrarán sólo a las variables libres de la expresión que se está analizando. El conjunto R, por su parte, dado que su contenido está intimamente relacionado con la función *sharerec*, puede contener variables que no aparezcan libres en dicha expresión, pero sí estén en ámbito.

Por otro lado, toda llamada a las reglas $\vdash_{check}$ realizada por el algoritmo de inferencia se realiza con variables que fueron anteriormente inferidas como n y fuerza que el tipo de las mismas sea d, r ó s.

Todas estas propiedades forman parte de los invariantes del algoritmo de inferencia, que se demostrarán a continuación:

LEMA 3. *Suponemos que durante el algoritmo de inferencia se obtiene* $e \vdash_{inf} (D, R, S, N)$ *y* $(D', R', S') \vdash_{check} e$ *para cierta expresión* e*. Las siguientes siete propiedades son ciertas:*

1. *D, R, S y N son disjuntos dos a dos.*
2. $D \cup S \cup N \subseteq FV(e)$, $R \subseteq scope(e)$ *y* $D \cup R \cup S \cup N \supseteq FV(e)$.
3. $\bigcup_{z \in D} sharerec(z, e) \subseteq D \cup R$
4. *D', R' y S' son disjuntos dos a dos.*
5. $D' \cup S' \subseteq N$, $R' \subseteq scope(e)$.
6. $\bigcup_{z \in D'} sharerec(z, e) \subseteq D' \cup R' \cup D$.
7. $R' \cap S = \emptyset$, $R' \cap D = \emptyset$.

Demostración.

Propiedades (1), (2) y (3)

Las tres primeras propiedades pueden demostrarse por inducción sobre e.

$\boxed{e = c}$ $\boxed{e = x}$ $\boxed{e = x@r}$ $\boxed{e = x!}$ $\boxed{e = C\ \overline{a_i}^n\ @r}$

Trivial, observando las reglas $\vdash_{inf}$ correspondientes. En el caso $e = x!$ se cumple $R \subseteq scope(e)$ debido a que la función *sharerec* sólo devuelve variables en ámbito en e. Además se tiene en este caso:

$$\begin{array}{rcl} \bigcup_{z \in D} sharerec(z, e) & = & sharerec(x, e) \\ & = & (sharerec(x, e) - \{x\}) \cup \{x\} \\ & = & R \cup D \end{array}$$

$\boxed{e = f\ \overline{a_i}^n\ @r}$

En este caso se obtiene:

$$\begin{array}{rcl} D & = & \bigcup_{i=1}^{n} D_i \\ R & = & \bigcup_{i=1}^{n} \{sharerec(a_i, f\overline{a_i}^n @r) - \{a_i\} \mid a_i \in D_i\} \\ S & = & \bigcup_{i=1}^{n} S_i \\ N & = & \bigcup_{i=1}^{n} N_i - \bigcup_{i=1}^{n} S_i \end{array}$$

Se cumple $D \cap R = \emptyset$, $D \cap S = \emptyset$, $D \cap N = \emptyset$, $R \cap S = \emptyset$, $R \cap N = \emptyset$ por hipótesis en la regla [APP$_I$] y debido al hecho de que I_D, I_S e I_N son disjuntos. Por otro lado, la igualdad $S \cap N = \emptyset$ resulta evidente.

Con respecto a la propiedad (2) se verifica $R \subseteq scope(e)$, por definición de *sharerec*. El resto de inclusiones se cumplen ya que $D \cup S \cup N = FV(e)$. Supongamos $x \in D$. Se tiene $x \in D_i$ para algún i y, por tanto, x es un parámetro en la llamada de la función. Los casos $x \in S$ y $x \in N$ son análogos. Por otro lado, si $x \in FV(e)$ entonces x está contenido en D_i, S_i o N_i, ya que $I_D \cup I_S \cup I_N = \{1..n\}$.

Para demostrar la propiedad (3) consideramos $y \in sharerec(z, e)$ para algún $z \in \bigcup_{i=1}^{n} D_i$. Esto implica que existe un $i \in \{1..n\}$ tal que $y \in sharerec(a_i, e)$ y $a_i \in D_i$. Distinguimos casos:

- Si $y = a_i$, entonces $y \in D$.
- Si $y \neq a_i$, entonces $y \in R$.

$\boxed{e = \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}$

Los conjuntos inferidos son:

$$\begin{array}{rcl} D & = & (D_1 \cup D_2) - \{x_1\} \\ R & = & R_1 \cup (R_2 - D_1) \\ S & = & ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2) \\ N & = & ((N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2) - \{x_1\} \end{array}$$

Por hipótesis de inducción sabemos que D_1, R_1, S_1 y N_1 son disjuntos dos a dos y que D_2, R_2, S_2, N_2 también lo son.

Puede demostrarse $D \cap R = \emptyset$ a partir de la hipótesis de inducción y a partir del hecho de que $R_1 \cap D_2 = \emptyset$, el cual queda garantizado por el hecho de que el operador $\triangleright$ de la regla [LET$_I$] esté definido. En efecto, este operador impide variables comunes en $FV(e_2)$ y R_1 y por hipótesis de inducción aplicada a la proposición (2) se cumple $D_2 \subseteq FV(e_2)$.

La demostración de $D \cap S = \emptyset$ es similar a la anterior. Tan sólo es necesario demostrar $D_1 \cap S_2 = \emptyset$, que de nuevo se deduce a partir de la definición de $\triangleright$ y que $S_2 \subseteq FV(e)$. Para el resto de pares de conjuntos la demostración es análoga.

Para la propiedad (2) puede verse fácilmente que $D \cup S \cup N \subseteq FV(e)$:

$$\begin{array}{rcl} FV(e) & = & (FV(e_1) \cup FV(e_2)) - \{x_1\} \\ & = & (FV(e_1) - \{x_1\}) \cup (FV(e_2) - \{x_1\}) \\ & \supseteq & ((D_1 \cup S_1 \cup N_1) - \{x_1\}) \cup ((D_2 \cup S_2 \cup N_2) - \{x_1\}) \\ & = & ((D_1 \cup D_2) - \{x_1\}) \cup ((S_1 \cup S_2) - \{x_1\}) \cup ((N_1 \cup N_2) - \{x_1\}) \\ & \supseteq & D \cup S \cup N \end{array}$$

Además, sea $x \in R$. Según la definición de R pueden distinguirse dos casos:

- $x \in R_1$: Por la hipótesis de inducción sabemos que $R_1 \subseteq scope(e_1)$. Por tanto, $x \in scope(e_1) = scope(e)$.
- $x \in R_2$: En este caso tenemos $x \in scope(e_2) = scope(e) \cup \{x_1\}$. Sin embargo, se cumple que $x \neq x_1$, ya que en caso contrario tendríamos $x_1 \in R_2$ y no podríamos aplicar la regla [LET$_I$], puesto que el conjunto C no estaría definido. Por tanto, $x \in scope(e)$.

Con esto queda demostrado que $R \subseteq scope(e)$. Queda mostrar que $D \cup R \cup S \cup N \supseteq FV(e)$. Para ello suponemos $x \in FV(e)$; sabemos, por tanto, que $x \neq x_1$. Distinguimos casos:

- $x \in FV(e_1)$: Por hipótesis de inducción tenemos $x \in D_1 \cup R_1 \cup S_1 \cup N_1$.
  - Si $x \in D_1$ entonces $x \in D$.
  - Si $x \in R_1$ entonces $x \in R$.
  - Si $x \in S_1$, $x \notin N_2$ y $x \notin D_2 \cup R_2$ entonces $x \in S$.
  - Si $x \in S_1$ y $x \in N_2$ entonces $x \in N$.
  - Si $x \in S_1$ pero $x \in D_2 \cup R_2$ se tiene $x \in R$ o $x \in D$. La demostración es similar al caso en que $x \in FV(e_2)$.
  - Si $x \in N_1$ y $x \notin (D_2 \cup R_2 \cup S_2)$ entonces $x \in N$.
  - Si $x \in N_1$ y $x \in (D_2 \cup R_2 \cup S_2)$, aunque no se cumple necesariamente que $x \in FV(e_2)$, cuyo caso se verá a continuación, la demostración es similar a dicho caso, ya que $x \neq x_1$.
- $x \in FV(e_2)$: Por hipótesis de inducción tenemos $x \in D_2 \cup R_2 \cup S_2 \cup N_2$.
  - Si $x \in D_2$ entonces $x \in D$.
  - Si $x \in R_2$ y $x \notin D_1$ entonces $x \in R$.
  - Si $x \in R_2$ y $x \in D_1$ entonces $x \in D$.
  - Si $x \in S_2$ entonces $x \in S$.
  - Si $x \in N_2$ entonces $x \in N$.

Reuniendo todos los casos puede verse que $x \in D \vee x \in R \vee x \in S \vee x \in N$. Por tanto, $x \in D \cup R \cup S \cup N$.

A continuación se demostrará la propiedad (3). Sea $y \in \mathit{sharerec}(z, e)$ para algún $z \in (D_1 \cup D_2) - \{x_1\}$. Obviamente se cumple $y \neq x_1$, ya que $x_1 \notin \mathit{scope}(e)$. Distinguimos casos según la procedencia de z:

- $z \in D_1$

  Dado que $\mathit{scope}(e_1) = \mathit{scope}(e)$ se tiene $y \in \mathit{sharerec}(z, e_1)$ para algún $z \in D_1$. La hipótesis de inducción establece que:

  $$\bigcup_{z \in D_1} \mathit{sharerec}(z, e_1) \subseteq D_1 \cup R_1$$

  Por tanto, $y \in D_1 \cup R_1$. Si $y \in D_1$ entonces $y \in D$. Si $y \in R_1$ entonces $y \in R$.

- $z \in D_2$

  En este caso $\mathit{scope}(e_2) = \mathit{scope}(e) \cup \{x_1\}$. Sin embargo también se cumple $y \in \mathit{sharerec}(z, e_2)$ para algún $z \in D_2$, ya que $y \neq x_1$. De nuevo puede aplicarse la hipótesis de inducción:

  $$\bigcup_{z \in D_2} \mathit{sharerec}(z, e_2) \subseteq D_2 \cup R_2$$

  De este modo se tiene $y \in D_2 \cup R_2$. Si $y \in D_2$ entonces $y \in D$. Si $y \in R_2$ entonces se cumple $y \in R$, o bien $y \in D$.

$\boxed{e = \textbf{case}\ x\ \textbf{of}\ \overline{C\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}}$

Se infieren los siguientes conjuntos:

$$\begin{array}{rcl} D & = & \bigcup_{i=1}^{n}(D_i - P_i) \\ R & = & \bigcup_{i=1}^{n}(R_i - P_i) \\ S & = & \bigcup_{i=1}^{n}(S_i - P_i) \\ N' & = & (\bigcup_{i=1}^{n}(N_i - P_i)) - (D \cup R \cup S) \\ N & = & \begin{cases} N' \cup \{x\} & \text{si } x \notin D \cup R \cup S \\ N' & \text{e.o.c.} \end{cases} \end{array}$$

Para la propiedad (1) comenzamos demostrando que $D \cap R = \emptyset$ por reducción al absurdo. Supongamos que existe una variable z tal que $z \in D$ y $z \in R$. Por tanto, existen $i, j \in \{1..n\}$ tal que $z \in (D_i - P_i)$ y $z \in (R_j - P_j)$.

- Si $i = j$ sabemos que $z \in D_i$ y $z \in R_i$, lo cual contradice el hecho de que D_i y R_i son disjuntos, establecido por la hipótesis de inducción.

- Si $i \neq j$ tenemos que $(D_i - P_i) \cap (R_j - P_j) \neq \emptyset$, lo cual contradice el hecho de que el operador $\sqcup$ esté bien definido.

La demostración de $D \cap S = \emptyset$ y $R \cap S = \emptyset$ es similar. Por otro lado resulta obvio que N' es disjunto de D, R y S. Distinguiendo casos sobre $z = x$ o $z \neq x$ puede también establecerse fácilmente que N es disjunto de D, R y S.

A continuación demostramos la propiedad (2). En primer lugar:

$$\begin{array}{rcl} FV(e) & = & \bigcup_{i=1}^{n} \left(FV(e_i) - \{x_{ij} | j \in \{1..n_i\}\}\right) \cup \{x\} \\ & = & \bigcup_{i=1}^{n} \left(FV(e_i) - P_i\right) \cup \{x\} \\ & \supseteq & \bigcup_{i=1}^{n} \left((D_i \cup S_i \cup N_i) - P_i\right) \cup \{x\} \\ & = & \bigcup_{i=1}^{n} (D_i - P_i) \cup \bigcup_{i=1}^{n} (S_i - P_i) \cup \bigcup_{i=1}^{n} (N_i - P_i) \cup \{x\} \\ & \supseteq & D \cup S \cup N' \cup \{x\} \\ & \supseteq & D \cup S \cup N \end{array}$$

Por otro lado, supongamos $z \in R$. En tal caso existe un $i \in \{1..n\}$ tal que $z \in R_i$ y $z \notin P_i$. Tenemos:

$$z \in R_i \Rightarrow z \in \mathit{scope}(e_i) \Rightarrow z \in \mathit{scope}(e)$$

El primer paso se obtiene por hipótesis de inducción y el último paso se deduce a partir del hecho de que $\mathit{scope}(e_i) = \mathit{scope}(e) \cup P_i$ y que $z \notin P_i$.

Para demostrar que $D \cup R \cup S \cup N \supseteq FV(e)$ tomamos $z \in FV(e)$.

- Si $z = x$ entonces, o bien $z \in N$, o bien $z \in D \cup R \cup S$.
- Si $z \neq x$ entonces $z \in FV(e_i)$ y $z \notin P_i$ para algún $i \in \{1..n\}$. Por hipótesis de inducción tenemos $z \in D_i \cup R_i \cup S_i \cup N_i$.
  - Si $z \in D_i$ (resp. R_i, S_i), tenemos que $z \in D$ (resp. R, S), ya que $z \notin P_i$.
  - Si $z \in N_i$ entonces, o bien $z \in N$, o bien $z \in D \cup R \cup S$.

Con respecto a la propiedad (3) sea $y \in \mathit{sharerec}(z, e)$ para algún $z \in D$. Dado que ninguno de los P_i se encuentra en $\mathit{scope}(e)$ se tiene $y \notin P_i$ para todo $i \in \{1..n\}$. Por otro lado $z \in D_j$ y $z \notin P_j$ para algún $j \in \{1..n\}$. La hipótesis de inducción establece que:

$$\forall i \in \{1..n\} \,.\, \bigcup_{z \in D_i} \mathit{sharerec}(z, e_i) \subseteq D_i \cup R_i$$

Como se cumple $\mathit{scope}(e) \subseteq \mathit{scope}(e_j)$, se tiene $y \in \mathit{sharerec}(z, e_j)$ para algún $z \in D_j$. Esto implica $y \in D_j \cup R_j$. Si $y \in D_j$ se tiene $y \in D$, ya que $y \notin P_j$. Si $y \in R_j$ se tiene, de forma análoga, $y \in R$.

$$\boxed{e = \textbf{case}!\ x\ \textbf{of}\ \overline{C\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}}$$

Se obtienen los siguientes cuatro conjuntos:

$$\begin{array}{rcl} D & = & D' \cup \{x\} \\ R & = & (\mathit{sharerec}(x,e) \cup R') - \{x\} \\ S & = & S' \\ N & = & N' \\ & & \textbf{donde } (D',R',S',N') = \sqcup_{i=1}^{n}(D_i,R_i,S_i,N_i,P_i) \end{array}$$

Con un razonamiento similar al visto para el **case** se puede ver que D', R', S', N' son disjuntos dos a dos. Tan sólo es necesario demostrar:

$$x \notin S \qquad x \notin N \qquad \mathit{sharerec}(x,e) \cap D' = \emptyset \qquad \mathit{sharerec}(x,e) \cap S = \emptyset \qquad \mathit{sharerec}(x,e) \cap N = \emptyset$$

Todo esto puede deducirse a partir del uso del operador $\triangleright$, que establece que en las variables libres de las alternativas no pueden aparecer variables contenidas en $\mathit{sharerec}(x,e)$, ni la variable x. Si una variable pertenece a D', S o a N también pertenecerá a $FV(e)$ por la propiedad (2), que será demostrada a continuación.

Con un razonamiento análogo al del **case** se obtiene $FV(e) \supseteq D' \cup S' \cup N' \cup \{x\}$ y, por tanto, $FV(e) \supseteq D \cup S \cup N$.

Por otra parte, sea z una variable tal que $z \in R$:

- Si $z \in \mathit{sharerec}(x,e) - \{x\}$ entonces $z \in \mathit{scope}(e)$ por definición de la función *sharerec*, que sólo devuelve variables en ámbito.
- Si $z \in R'$ podemos seguir un razonamiento análogo al del **case** para deducir $z \in \mathit{scope}(e)$.

Con ello tenemos $R \subseteq \mathit{scope}(e)$. Sólo queda demostrar que $D \cup R \cup S \cup N \supseteq FV(e)$, para lo cual es aplicable en gran parte la explicación vista en el **case**. La única diferencia es que ahora la variable x está siempre contenida en D.

Con respecto a la propiedad (3), sea $y \in \mathit{sharerec}(z,e)$ para algún $z \in D$. Demostramos que $y \in D \cup R$: Si $y = x$ se tiene obviamente $y \in D$, por lo que en el resto de la demostración supondremos $y \neq x$. Distinguimos casos:

- Si $z = x$, entonces $y \in \mathit{sharerec}(x,e)$ y por tanto, $y \in R$.
- Si $z \neq x$ podemos seguir un razonamiento análogo al visto en el **case** no destructivo para obtener $y \in D' \cup R'$, a partir de lo cual se tiene $y \in D \cup R$.

Propiedades (4), (5), (6) y (7)

Estas cuatro propiedades están relacionadas con las reglas $\vdash_{\mathit{check}}$ y su relación con las correspondientes reglas $\vdash_{\mathit{inf}}$. Para verificar su validez demostraremos que se cumplen para cada llamada inicial a $\vdash_{\mathit{check}}$ y se conservan en cada llamada recursiva.

Llamada inicial en [LET$_I$]

Las propiedades (4), (6) y (7) se cumplen trivialmente, ya que $R' = D' = \emptyset$. Con respecto a (5) se tiene $R' = \emptyset \subseteq \mathit{scope}(e)$. Además, dado que $D' = \emptyset$ y en la definición

de S' se realiza intersección con $N_1 \subseteq N$ en el *check* sobre e_1 y con $N_2 \subseteq N$ en el *check* sobre e_2 se tiene $D' \cup S' \subseteq N$.

Llamada inicial en [CASE$_I$] Sea $e \equiv \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

Se realiza $\vdash_{check}$ en la i-ésima alternativa con los siguientes conjuntos:

$$\begin{array}{rcl} D' & = & (D \cup D'_i) \cap N_i \\ R' & = & R \cup ((R'_i \cup R''_i \cup R'''_i) - D_i) \\ S' & = & (S \cup S'_i) \cap N_i \end{array}$$

Demostramos (4) considerando cada par de conjuntos por separado:

- $D' \cap R' = \emptyset$.

  Ya se demostró en la propiedad (1) que $D \cap R = \emptyset$. También se cumple $D \cap (R'_i \cup R''_i) = \emptyset$, ya que R'_i sólo puede tener elementos de $Rec_i \subseteq P_i$, el conjunto R''_i sólo puede tener elementos de P_i, y D no contiene ningún elemento de P_i. Por otro lado, R'''_i no contiene elementos de N_i y $D' \subseteq N_i$, por lo que se tiene $R'''_i \cap D' = \emptyset$. Por último, $D'_i = \emptyset$ por lo que también es disjunto de R'.

- $D' \cap S' = \emptyset$

  Por la propiedad (1) se tiene $D \cap S = \emptyset$. Además D no contiene elementos de P_i y el conjunto S'_i sólo contiene variables contenidas en P_i, por lo que también se cumple $D \cap S'_i = \emptyset$. Además, dado que $D'_i = \emptyset$, también se tiene $D'_i \cap S' = \emptyset$.

- $R' \cap S' = \emptyset$

  Mediante un razonamiento análogo a los dos anteriores puede obtenerse $R \cap S' = \emptyset$. Por otro lado se cumple $S \cap (R'_i \cup R''_i) = \emptyset$, ya que $R'_i \cup R''_i$ sólo contiene elementos de P_i, que nunca estarán contenidos en S. Distinguiendo casos según $type(x)$ y a partir del hecho de que $Rec_i \cap (P_i - Rec_i) = \emptyset$ puede obtenerse $R'_i \cap S'_i = \emptyset$. Además $S' \subseteq N_i$ y el conjunto R'''_i no contiene ningún elemento de N_i, por lo que S' y R'''_i son disjuntos. Sólo queda demostrar que $R''_i \cap S'_i = \emptyset$, lo cual puede obtenerse a partir de una de las premisas de la regla [CASE$_I$].

La propiedad (5) es fácilmente demostrable a partir de las definiciones de D' y S', en las que se realiza intersección con el N_i correspondiente, que es subconjunto de N. Por otro lado se tiene $R \subseteq scope(e)$ a partir de (2) y, por tanto, $R \subseteq scope(e) \subseteq scope(e_i)$. Además se tiene:

$$\begin{array}{c} R'_i \subseteq Rec_i \subseteq scope(e_i) \\ R''_i \subseteq P_i \subseteq scope(e_i) \\ R'''_i \subseteq D \subseteq scope(e) \subseteq scope(e_i) \end{array}$$

Con ello se verifica $R' \subseteq scope(e_i)$.

A continuación demostramos (6). Para ello suponemos, para algún $i \in \{1..n\}$: $y \in sharerec(z, e_i)$, donde $z \in (D \cup D'_i) \cap N_i = D \cap N_i$. Tenemos que demostrar $y \in D' \cup R' \cup D_i$. Distinguimos casos:

- $y \in P_i$

  Se tiene $y \in R_i''$ y por tanto, o bien $y \in D_i$, o bien $y \in R'$.

- $y \notin P_i$.

  En este caso se tiene $y \in \mathit{sharerec}(z, e)$ para algún $z \in D$. La propiedad (3) permite establecer:

  $$\bigcup_{z \in D} \mathit{sharerec}(z, e) \subseteq D \cup R$$

  Por tanto, se cumple $y \in D \cup R$ y podemos realizar la siguiente distinción de casos:

  - Si $y \in D \ \wedge \ y \in N_i$, entonces $y \in D'$.
  - Si $y \in D \ \wedge \ y \notin N_i$, entonces $y \in R_i'''$, de lo que se obtiene $y \in R'$, o bien $y \in D_i$.
  - Si $y \in R$, entonces $y \in R'$.

Para demostrar (7) consideramos cada par de conjuntos por separado:

- $R' \cap D_i = \emptyset$

  Tan sólo es necesario demostrar $R \cap D_i = \emptyset$. Se hará por reducción al absurdo: Sea $x \in R \ \wedge \ x \in D_i$. Deberá existir un $j \in \{1..n\}$ tal que $x \in R_j$. Obviamente $x \notin P_j \ \wedge \ x \notin P_i$. Si $j = i$ se tiene $R_i \cap D_i \neq \emptyset$, lo cual contradice (1), ya que se ha obtenido $e_i \vdash_{inf} (D_i, R_i, S_i, N_i)$. Por el contrario, si $j \neq i$ se obtiene $x \in (R_j - P_j) \cap (D_i - P_i)$ lo cual contradice el hecho de que el operador $\sqcup$ está bien definido.

- $R' \cap S_i = \emptyset$

  Con un razonamiento similar al visto para $R \cap D_i = \emptyset$ se puede obtener $S \cap D_i = \emptyset$.

  Puede obtenerse $R_i' \cap S_i = \emptyset$ por reducción al absurdo: Sea $x \in R_i' \ \wedge \ x \in S_i$. Esto implica que $x \in \mathit{Rec}_i$ y que el discriminante del **case** tiene tipo d. Bajo estas condiciones se tiene $\mathit{Rec}_i \cap S_i \neq \emptyset$ y se contradice el hecho de que la relación *inh* esté bien definida.

  La igualdad $R_i'' \cap S_i = \emptyset$ se obtiene a partir de las premisas en la regla [CASE$_I$]. Por último R_i''' sólo contiene elementos de D. Con un razonamiento análogo al visto para $R \cap D_i = \emptyset$ puede demostrarse que $D \cap S_i = \emptyset$. Por tanto se verifica $R_i''' \cap S_i = \emptyset$.

Llamada inicial en [CASE!$_I$] Sea $e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

Se realiza una llamada a $\vdash_{check}$ en la i-ésima alternativa con los siguientes conjuntos:

$$\begin{array}{rcl} D' & = & (D^* \cup Rec_i) \cap N_i \\ R' & = & R^* \cup ((R'_i \cup R''_i \cup R'''_i) - D_i) \\ S' & = & (S^* \cup (P_i - Rec_i)) \cap N_i \\ & & \textbf{donde } (D^*, R^*, S^*, N^*) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \end{array}$$

Demostramos (4):

- $D' \cap R' = \emptyset$

  Ya se vió en la demostración de (1) que $D^* \cap R^* = \emptyset$. Además también se cumple $R^* \cap Rec_i = \emptyset$, ya que R^* no contiene ningún P_i. La igualdad $(R'_i \cup R''_i) \cap D' = \emptyset$ puede obtenerse a partir de las definiciones de R'_i y R''_i. Por último se tiene que R'''_i y D' son disjuntos, ya que el primero no contiene elementos de N_i y el segundo sólo contiene elementos de N_i.

- $D' \cap S' = \emptyset$

  Puede demostrarse fácilmente a partir del hecho de que $D^* \cap S^* = \emptyset$ y $Rec_i \cap (P_i - Rec_i) = \emptyset$.

- $R' \cap S' = \emptyset$

  Por un lado se verifica $R^* \cap S^* = \emptyset$ del un modo análogo al visto en la demostración de la propiedad (1). Por otra parte, R'_i y S^* también son disjuntos, ya que el primero sólo contiene elementos de P_i, que nunca aparecen en S^*. Las igualdades $R'_i \cap (P_i - Rec_i) = \emptyset$ y $R''_i \cap S^* = \emptyset$ pueden deducirse de las premisas de [CASE!$_I$]. Además R'''_i y S' son disjuntos, ya que el primero no contiene elementos de N_i y el segundo sólo contiene elementos de N_i.

  La demostración de $R''_i \cap (P_i - Rec_i) = \emptyset$ es algo más complicada: Si existiese una variable $x \in R''_i \cap (P_i - Rec_i)$ se daría la situación en la que un hijo no recursivo comparte un hijo recursivo de un hermano recursivo, lo cual no es posible.

La propiedad (5) se verifica fácilmente, ya que en la definición de D' y S' de cada alternativa se realiza intersección con los N_i correspondientes. Además se tiene:

$$\begin{array}{rcl} R^* \subseteq R \subseteq scope(e) \subseteq scope(e_i) & & \\ R'_i \subseteq scope(e_i) & & \text{(por definición de } \textit{sharerec}) \\ R''_i \subseteq scope(e_i) & & \text{(por definición de } \textit{sharerec}) \\ R'''_i \subseteq scope(e_i) & & \text{(por definición de } \textit{sharerec}) \end{array}$$

Por lo que $R' = R^* \cup ((R'_i \cup R''_i \cup R'''_i) - D_i) \subseteq scope(e_i)$.

Para verificar (6), sea $y \in sharerec(z, e_i)$ donde $z \in (D^* \cup Rec_i) \cap N_i$ para algún $i \in \{1..n\}$. Hemos de demostrar que $y \in D' \cup R' \cup D_i$. Si $y \in D_i$ es obvio que se verifica la pertenencia. Si $y \in (D^* \cup Rec_i) \cap N_i$ también se cumple $y \in D'$. Por tanto, supondremos en lo sucesivo que $y \notin ((D^* \cup Rec_i) \cap N_i) \cup D_i$.

Por otro lado, si se cumple $z \in Rec_i \cap N_i$ entonces se tiene que $y \in R''_i$ y, por tanto, $y \in R'$.

Supongamos que $z \in D_i \cap N_i$. Distinguimos casos:

- $y \in P_i$

  Se cumple $y \in R'_i$ y por tanto, $y \in R'$.

- $y \notin P_i$.

  Se tiene $y \in \mathit{sharerec}(z, e)$ para algún $z \in D^*$. Dado que se cumple la propiedad (3) para cada e_i, la siguiente inclusión puede deducirse a partir de la definición de $\sqcup$:

  $$\bigcup_{z \in D^*} \mathit{sharerec}(z, e) \subseteq D^* \cup R^*$$

  Por tanto, $y \in D^* \cup R^*$. Se distinguen casos:

  - Si $y \in R^*$ entonces $y \in R'$.
  - Si $y \in D^*$ e $y \in N_i$ entonces $y \in D'$.
  - Si $y \in D^*$ e $y \notin N_i$ se tiene $y \in R'''_i$ y, por tanto, $y \in R'$.

Demostramos la propiedad (7):

- $R' \cap D_i = \emptyset$

  Tan sólo es necesario demostrar $R^* \cap D_i = \emptyset$. Para ello puede seguirse un razonamiento análogo al visto en el **case** no destructivo.

- $R' \cap S_i = \emptyset$

  Siguiendo un razonamiento análogo al visto en el **case** no destructivo puede deducirse $R^* \cap S_i = \emptyset$. Por otro lado, la igualdad $(R'_i \cup R''_i) \cap S_i = \emptyset$ se obtiene a partir de las premisas en [CASE!$_I$]. Por último, dado que $D^* \cap S_i = \emptyset$ y que R'''_i sólo contiene elementos de D^*, se obtiene $R'''_i \cap S_i = \emptyset$

Llamada recursiva en [LET$_C$] Sea $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$

Supongamos que la llamada $(D_p, R_p, S_p) \vdash_{\mathit{check}} e$ cumple las propiedades (4), (5), (6) y (7). Existen dos llamadas a $\vdash_{\mathit{check}}$. La primera de ellas se realiza sobre e_1 con los siguientes conjuntos:

$$\begin{array}{rcl} D' & = & D_p \cap N_1 \\ R' & = & R_p - D_1 \\ S' & = & S_p \cap N_1 \end{array}$$

A partir del hecho de que D_p, R_p y S_p son disjuntos dos a dos puede verse que $D_p \cap N_1$, $R_p - D_1$ y $S_p \cap N_1$ también lo son, por lo que la propiedad (4) se verifica.

La propiedad (5) también se cumple, ya que en D' y S' se realiza intersección con N_1. Además se tiene $R' \subseteq R_p \subseteq \mathit{scope}(e) \subseteq \mathit{scope}(e_1)$.

Con respecto a (6), supongamos que $y \in \mathit{sharerec}(z, e_1)$ para algún $z \in D_p \cap N_1$. Tenemos que demostrar:

$$y \in (D_p \cap N_1) \cup (R_p - D_1) \cup D_1 = D' \cup R' \cup D_1$$

Obviamente tenemos $y \neq x_1$, ya que $x_1 \notin \mathit{scope}(e_1)$. Además, debido a que $\mathit{scope}(e_1) = \mathit{scope}(e)$ se tiene $y \in \mathit{sharerec}(z, e)$ para algún $z \in D_p$. La hipótesis de inducción establece que:

$$\bigcup_{z \in D_p} \mathit{sharerec}(z, e) \subseteq D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$$

Por tanto, $y \in D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$. Distinguimos casos:

- $y \in D_p$

  Por la propiedad (5) se tiene $y \in N \subseteq N_1 \cup N_2$.

  - Si $y \in N_1$ entonces $y \in D_p \cap N_1$ y, por tanto, $y \in D'$.
  - Si $y \in N_2$ entonces se tendría $y \in FV(e_2)$ e $y \in R''_p$, lo cual está prohibido por el operador $\triangleright$ de la regla [LET$_C$].

- $y \in R_p$

  En este caso se tiene $y \in D_1$, o bien $y \in R'$.

- $y \in (D_1 \cup D_2) - \{x_1\}$

  Se ha visto que $y \neq x_1$, con lo que $y \in D_1 \cup D_2$. Si $y \in D_1$ la propiedad se cumple trivialmente. Si $y \in D_2$ se tendría $y \in FV(e_2)$ e $y \in R''_p$, lo cual no es posible debido al uso del operador $\triangleright$ en la regla [LET$_C$].

Para demostrar la propiedad (7), se tiene trivialmente que $(R_p - D_1) \cap D_1 = \emptyset$. Por otro lado $R_p \cap S_1 = \emptyset$ debido a una de las premisas en [LET$_C$].

Consideremos ahora la llamada recursiva a $\vdash_{\mathit{check}}$ realizada sobre e_2:

$$\begin{array}{rcl} D' & = & D_p \cap N_2 \\ R' & = & (R_p \cup R'_p) - D_2 \\ S' & = & S_p \cap N_2 \end{array}$$

Para demostrar la propiedad (4) se sigue un razonamiento análogo al visto en la llamada a $\vdash_{\mathit{check}}$ sobre e_1. Tan sólo hay que demostrar dos igualdades adicionales $R'_p \cap D' = \emptyset$ y $R'_p \cap S' = \emptyset$, que se deducen a partir del hecho de que R'_p no contiene elementos de P_i y que los conjuntos D' y S' sólo contienen elementos de P_i.

De un modo similar al visto en la llamada a $\vdash_{\mathit{check}}$ sobre e_1, se verifica también en este caso (5).

Para demostrar (6) supongamos que $y \in \mathit{sharerec}(z, e_2)$ para algún $z \in D_p \cap N_2$. Hemos de demostrar:

$$y \in (D_p \cap N_2) \cup ((R_p \cup R'_p) - D_2) \cup D_2 = D' \cup R' \cup D_2$$

Si $y = x_1$, una de las premisas de [LET$_C$] obliga a que $x \notin S_2 \cup N_2$. La única posibilidad es $y \in D_2$, ya que en caso contrario el conjunto C no estaría definido. Supondremos ahora que $y \neq x_1$. En este caso se tiene $y \in \mathit{sharerec}(z, e)$, por lo que podemos aplicar la hipótesis de inducción del mismo modo que en la llamada a $\vdash_{\mathit{check}}$ sobre e_1. Esto nos lleva a $y \in D_p \cup R_p \cup ((D_1 \cup D_2) - \{x_1\})$. Distinguimos casos:

- $y \in D_p$
  - Si $y \in N_2$ entonces $y \in D'$.
  - Si $y \in N_1 - N_2$ entonces $y \in R'_p$ y por tanto, $y \in D_2 \ \vee \ y \in R'$
- $y \in R_p$

  Se tiene $y \in D_2 \ \vee \ y \in R'$.
- $y \in (D_1 \cup D_2) - \{x_1\}$

  Si $y \in D_2$ se verifica la propiedad. Si $y \in D_1 - D_2$ entonces $y \in R'_p$ y por tanto, $y \in R'$.

A continuación demostramos (7):

- $((R_p \cup R'_p) - D_2) \cap D_2 = \emptyset$

  Trivialmente es cierto.
- $((R_p \cup R'_p) - D_2) \cap S_2 = \emptyset$

  Se cumple $R_p \cap S_2 = \emptyset$ a partir de una de las premisas en [LET$_C$]. Para obtener $R'_p \cap S_2 = \emptyset$ razonamos por reducción al absurdo: Supongamos que $x \in R'_p \ \wedge \ x \in S_2$. Entonces $x \in FV(e_2)$ por la propiedad (2). Además, por definición de R'_p, se tiene $x \in (D_p \cap N_1)$, o bien $x \in D_1$. En el primer caso se contradice el uso de $\triangleright$ en la regla [LET$_C$]. En el segundo caso se contradice el uso de $\triangleright$ en la regla [LET$_I$].

Llamada recursiva en [CASE$_C$] Sea $e \equiv \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

Suponemos que la llamada $(D_p, R_p, S_p) \vdash_{check} e$ cumple las cuatro propiedades que queremos demostrar. Para cada i-ésima llamada recursiva los conjuntos son:

$$\begin{array}{rcl} D' & = & (D_p \cup D_{p_i}) \cap N_i \\ R' & = & (R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i \\ S' & = & (S_p \cup S_{p_i}) \cap N_i \end{array}$$

A partir de que D_p y S_p son disjuntos y $D_{p_i} = \emptyset$ puede deducirse que D' y S' también lo son. Para demostrar la propiedad (4) sólo queda obtener que R' es disjunto con D' y S'. Demostraremos que $R' \cap S' = \emptyset$ (la igualdad $R' \cap D' = \emptyset$ es análoga). Por hipótesis de inducción tenemos $R_p \cap S_p = \emptyset$. También se tiene $(R_{p_i} \cup R'_{p_i}) \cap S_p = \emptyset$, ya que S_p sólo tiene variables libres en e y $R_{p_i} \cup R'_{p_i}$ sólo tiene elementos de P_i. Distinguiendo casos sobre el conjunto en el que esté contenido el discriminante de **case** puede obtenerse $R_{p_i} \cap S_{p_i} = \emptyset$ en cualquier caso. Además $R'_{p_i} \cap S_{p_i} = \emptyset$ por hipótesis en [CASE$_C$]. Por último se verifica $R''_{p_i} \cap S' = \emptyset$, ya que R''_{p_i} no contiene elementos de N_i. Reuniendo todas estas igualdades se obtiene:

$$((R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i) \cap ((S_p \cup S_{p_i}) \cap N_i) = \emptyset$$

Con lo que se verifica que R' y S' son disjuntos.

La demostración de (5) es, al igual que en los demás casos, bastante sencilla. Se cumple $D' \cup S' \subseteq N_i$ a partir del hecho de que en las definiciones de D' y S' se realiza intersección con el mismo N_i. Por otro lado se tiene $R_p \subseteq scope(e) \subseteq scope(e_i)$. R'_{p_i} sólo tiene elementos de P_i, que también se encuentran en $scope(e_i)$. Además se cumple $R'_{p_i} \cup R''_{p_i} \subseteq scope(e_i)$ ya que tanto R'_{p_i} como R''_{p_i} sólo contienen elementos del resultado de *sharerec* aplicado a e_i.

Para la propiedad (6) suponemos $y \in sharerec(z, e_i)$, para alguna variable $z \in (D_p \cup D_{p_i}) \cap N_i$. Como $D_{p_i} = \emptyset$ se tiene $z \in D_p$. Hemos de demostrar:

$$y \in (((D_p \cup D_{p_i}) \cap N_i) \cup ((R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i) \cup D_i = D' \cup R' \cup D_i$$

Si $y \in P_i$ se tiene $y \in R'_{p_i}$ y por tanto se cumple $y \in D_i \cup R'$.

Si $y \notin P_i$, entonces $y \in sharerec(z, e)$ para algún $z \in D_p$. La hipótesis de inducción establece que:

$$\bigcup_{z \in D_p} sharerec(z, e) \subseteq D_p \cup R_p \cup D$$

Por tanto, $y \in D_p \cup R_p \cup D$. Distinguimos casos:

- $y \in D_p$

  Si $y \in N_i$ entonces $y \in D'$. Si $y \notin N_i$ se tiene $y \in R''_{p_i}$, a partir de lo cual puede deducirse $y \in D_i \cup R'$.

- $y \in R_p$

  En este caso se cumple siempre $y \in D_i \cup R'$.

- $y \in D$

  Si $y \notin N_i$ entonces $y \in R''_{p_i}$. Por tanto, $y \in D_i \cup R'$. Si $y \in N_i$ entonces $y \in D_p$ por el *check* de la regla [CASE$_I$].

A continuación demostramos (7). La igualdad es $R' \cap D_i = \emptyset$ es evidente a partir de la definición de R'. Pasamos a demostrar $R' \cap S_i = \emptyset$. Se tiene $(R_p \cup R'_{p_i}) \cap S_i = \emptyset$ por la premisa en regla [CASE$_C$]. Por otro lado, sea z una variable tal que $z \in R_{p_i}$ y $z \in S_i$. La primera pertenencia obliga a que $x \in D_p$, donde x es el discriminante del **case**. Además se tiene $z \in Rec_i \cap S_i$, lo que contradice el uso del predicado *inh* de la regla [CASE$_C$]. Por tanto, $R_{p_i} \cap S_i = \emptyset$. Por último se tiene $R''_{p_i} \cap S_i = \emptyset$, ya que R''_{p_i} sólo contiene variables de D y N y se cumple que $D \cap S_i = N \cap S_i = \emptyset$ por definición de $\sqcup$.

Llamada recursiva en [CASE!$_C$] Sea $e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

La demostración para este caso es una versión simplificada de la vista en la llamada recursiva en [CASE$_C$]. Por tanto, no se repetirá aquí. □

Otro detalle importante de cara a la demostración de corrección del algoritmo es la comprobación del hecho de que toda variable va a adquirir tipo d, r ó s al final del algoritmo, bien sea mediante las reglas $\vdash_{inf}$ o bien sea mediante un $\vdash_{check}$ posterior. Esto puede demostrarse distinguiendo casos en función de si una variable es libre o ligada y, en este último caso, en qué expresión ha sido introducida. De este modo es posible comprobar que en casi todos los casos dicha variable habrá adquirido marca d, r o s en algún momento del algoritmo. A continuación puede encontrarse la demostración formal de esta proposición. Nótese que ésta supone la existencia de un punto fijo en el cálculo de la signatura, cosa que será demostrada en la Sección 6.4.

LEMA 4. *Sea $f\ \overline{x_i}^n = e$ una definición. Si el algoritmo de inferencia consigue finalmente $e \vdash_{inf} (D, \emptyset, S, N)$, incluyendo el cálculo del punto fijo y una vez realizado el $\vdash_{check}$ final que obliga que todas las variables de N obtengan tipo s, entonces para toda $x \in var(e)$, x tiene una marca d, s ó r.*

Demostración. Por inducción sobre la profundidad en la que x pasa a estar en ámbito.

- **Base:** Si $x \in scope(e)$ entonces x es uno de los parámetros x_i, es decir, una de las variables libres en e. Toda variable que pertenezca a N acaba obteniendo marca s, debido al mphcheck final realizado:

$$(\emptyset, \emptyset, N) \vdash_{check} e$$

- **Paso inductivo:** Si $x \in var(e)$ pero $x \notin scope(e)$ entonces x es una variable ligada.

  Si x fue introducida en una expresión **let** $x_1 = e_1$ **in** e_2 y fue inferida con marca n (es decir, $x_1 \in N_2$), el siguiente $\vdash_{check}$ de la regla [LET$_I$] obligó a que adquiriese marca s:

$$(\emptyset, \emptyset, (S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2$$

  Si x fue introducida como variable patrón en una expresión **case**! x' **of** $\overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^n$ entonces es uno de los x_{ij} para algún $i \in \{1..n\}$ y algún $j \in \{1..n_i\}$. En tal caso se aplica el siguiente mphcheck sobre e_i:

$$((D \cup Rec_i) \cap N_i, R' \cap N_i, (S \cup (P_i - Rec_i)) \cap N_i) \vdash_{check} e_i$$

  Resulta obvio que si $x = x_{ij}$ para algún $i \in \{1..n\}$ y algún $j \in \{1..n_i\}$, entonces $x \in Rec_i$ o bien $x \in P_i - Rec_i$. En el primer caso x obtendrá tipo d y en el segundo obtendrá tipo s.

  Si x fue introducida en una expresión de la forma **case** x' **of** $\overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^n$ tenemos que x también es uno de los x_{ij} que entran en ámbito en cierta alternativa e_i. Como x' ha entrado en ámbito en un contexto superior, por hipótesis de inducción ya le ha sido asignada marca d, r o s. Si x' tiene marca s se realiza el siguiente *check* sobre e_i:

$$(D \cap N_i, R \cap N_i, (S \cup P_i) \cap N_i) \vdash_{check} e_i$$

por lo que x obtiene marca s. Si x' tiene marca d estamos en una situación similar al **case**!: si x se encuentra en posición recursiva del C_i correspondiente adquiere tipo r y en caso contrario adquiere tipo s. Por último, si x' tiene marca r, entonces x obtiene marca s

□

6.3. Corrección del algoritmo

Consideramos una definición $f\ \overline{x_i}^n = e$ que ha sido bien tipada por el algoritmo de inferencia. Para cada subexpresión e' de e se han inferido cuatro conjuntos mediante las reglas de $\vdash_{inf}$. Esta inferencia se ha realizado una única vez para cada subexpresión:

$$e' \vdash_{inf} (D, R, S, N)$$

Por otro lado ya se ha visto que las reglas de $\vdash_{inf}$ pueden provocar en determinados nodos del árbol abstracto un recorrido descendente dirigido por las reglas $\vdash_{check}$. De hecho una misma expresión puede sufrir varios $\vdash_{check}$ a lo largo del algoritmo, cada vez con un conjunto distinto de variables:

$$\begin{array}{rcl} (D_1, R_1, S_1) & \vdash_{check} & e' \\ & \vdots & \\ (D_m, R_m, S_m) & \vdash_{check} & e' \end{array}$$

Utilizaremos la notación $\vdash^*_{check}$ para hacer referencia a la reunión de todas las variables con las que se ha hecho $\vdash_{check}$ sobre una expresión. Es decir:

$$(D', R', S') \vdash^*_{check} e \textbf{ donde } D' = \bigcup_{i=1}^{m} D_i \qquad R' = \bigcup_{i=1}^{m} R_i \qquad S' = \bigcup_{i=1}^{m} S_i$$

Por el Lema 4 sabemos que toda variable inferida inicialmente como n adquirirá otra marca distinta a lo largo del algoritmo en un $\vdash_{check}$. Podemos afirmar que para toda subexpresión e' de e se cumple $N \subseteq D' \cup R' \cup S'$.

El objetivo de esta sección es demostrar la corrección del algoritmo estableciendo una correspondencia entre las reglas $\vdash_{inf}$ y $\vdash_{check}$ con las reglas del sistema de tipos. Se mostrará que si el algoritmo se ejecuta correctamente para una definición es posible tipar dicha definición con el sistema de tipos. Además se establece una relación entre los conjuntos D, R y S obtenidos por el algoritmo y el entorno Γ necesario para tipar la definición.

TEOREMA 1. *Supongamos que la declaración de función* ***no recursiva*** $f\ \overline{x_i}^n = e$ *ha sido aceptada por el algoritmo de inferencia (incl. el check final) y sea e' cualquier subexpresión de e para la cual el algoritmo ha obtenido $e' \vdash_{inf} (D, R, S, N)$ y $(D', R', S') \vdash^*_{check} e'$. Entonces existe un tipo seguro s y un entorno Γ^P que cumplen:*

1. $\Gamma^P \vdash e' : s$
2. $\forall x \in scope(e') :$
   a) $\Gamma^P(x) = d \Leftrightarrow x \in D \cup D'$
   b) $\Gamma^P(x) = s \Leftrightarrow x \in S \cup S'$
   c) $\Gamma^P(x) = r \Leftrightarrow x \in R \cup R'$

Demostración. Por inducción estructural sobre e'.

$\boxed{e' = c}$

$$\frac{}{\emptyset \vdash c : B}\ [\text{LIT}] \qquad \frac{}{c \vdash_{inf} (\emptyset, \emptyset, \emptyset, \emptyset)}\ [\text{LIT}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} c}\ [\text{LIT}_C]$$

Sea $\Gamma^P = [y : r \mid y \in R']$. El entorno $\emptyset$ verifica $\emptyset \vdash e' : s$. Mediante el uso de la regla [EXTS] puede obtenerse $\Gamma^P \vdash e' : s$, por lo que (1) se cumple. Por otro lado también se verifica la propiedad (2), ya que $D \cup D' = S \cup S' = \emptyset$ y $R \cup R' = R'$.

$\boxed{e' = x}$

$$\frac{}{[x : s]^P \vdash x : s}\ [\text{VAR}] \qquad \frac{}{x \vdash_{inf} (\emptyset, \emptyset, \{x\}, \emptyset)}\ [\text{VAR}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x}\ [\text{VAR}_C]$$

Tomamos $\Gamma^P = [x : s] + [y : r \mid y \in R']$. Está bien definido ya que $S = \{x\}$ y según el Lema 3 (7) los conjuntos S y R' son disjuntos. El entorno $[x : s]$ cumple $[x : s] \vdash e' : s$. A partir de esto y utilizando la regla [EXTS] puede obtenerse $\Gamma^P \vdash e' : s$. Por otra parte, la propiedad (2) también se verifica en este caso, ya que $D \cup D' = \emptyset$, $S \cup S' = \{x\}$ y $R \cup R' = R'$.

$\boxed{e' = x!}$

$$\frac{\begin{array}{c} R = sharerec(x, x!) - \{x\} \\ \Gamma_R = \{y : danger(type(y)) \mid y \in R\} \end{array}}{\Gamma_R + [x : T!@\rho]^P \vdash x! : T@\rho}\ [\text{REUSE}] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x!}\ [\text{REUSE}_C]$$

$$\frac{R = sharerec(x, x!) - \{x\} \quad type(x) = T@\rho}{x! \vdash_{inf} (\{x\}, R, \emptyset, \emptyset)}\ [\text{REUSE}_I]$$

Sea $\Gamma^P = [x : d] + [y : r \mid y \in R \cup R']$. Está bien definido ya que $D = \{x\}$ y por el Lema 3 (1,7) se cumple $R \cap D = \emptyset$ y $R' \cap D = \emptyset$, ya que $D = \{x\}$.

Sea $\Gamma'^P = [x : d] + [y : r \mid y \in R]$. Dado que $R = sharerec(x, e') - \{x\}$ se tiene $\Gamma'^P \vdash e' : s$. Mediante repetidas aplicaciones de la regla [EXTS] puede obtenerse $\Gamma^P \vdash e' : s$.

La propiedad (2) resulta trivial, ya que $D \cup D' = \{x\}$ y $S \cup S' = \emptyset$.

$\boxed{e' = x@r}$

$$\frac{\Gamma_1^P \geq_{x@r} [x : T@\rho', r : \rho] \quad \rho \neq \rho'}{\Gamma_1^P \vdash x@r : T\ @\rho}\ [\text{COPY}] \qquad \frac{}{x@r \vdash_{inf} (\emptyset, \emptyset, \emptyset, \{x\})}\ [\text{COPY}_I]$$

En este caso $x \in N$, y dado que $N \subseteq D' \cup R' \cup S'$ se realiza distinción de casos:

- $x \in D'$

  Sobre esta expresión se ha realizado check mediante la regla [COPY1$_C$]. Se toma $\Gamma^P = [x : d] + [y : r \mid y \in R'] + [r : \rho]$. Este entorno está bien definido por el Lema 3 (7). Puede demostrarse (1) a partir del hecho de que $\Gamma^P \geq_{e'} [x : s, r : \rho]$. En efecto, la única variable con tipo d en Γ^P es x, que pertenece a D'. Por el Lema 3 (6) se tiene que todas las variables contenidas en $sharerec(x, e')$ pertenecen a $D' \cup R' \cup D = \{x\} \cup R'$. Dado que todas las variables contenidas en $\{x\} \cup R'$ tienen tipo inseguro en Γ^P, se cumplen una de las tres condiciones que impone la relación $\geq_{e'}$. Las otras dos condiciones se cumplen trivialmente y, por tanto, puede obtenerse $\Gamma^P \vdash e' : s$.

  Por otro lado, se tiene $D \cup D' = \{x\}$, $R \cup R' = R'$ y $S \cup S' = \emptyset$ por lo que también se verifica la propiedad (2).

- $x \in R'$

  En este caso el check realizado corresponde a la regla [COPY2$_C$]. Sea $\Gamma^P = [y : r \mid y \in R'] + [r : \rho]$. Se cumple que $\Gamma^P \geq_{e'} [x : s, r : \rho]$, ya que $x \in dom(\Gamma^P)$ y se tiene $\Gamma^P(x) = r \geq s$. Además se cumple la tercera propiedad de $\geq_{e'}$, puesto que no hay variables con tipo d en Γ^P. Por tanto puede derivarse $\Gamma^P \vdash e' : s$.

  Las propiedad (2) puede deducirse del hecho de que $D \cup D' = S \cup S' = \emptyset$ y en Γ^P no existen variables con tipo s ó d. Además $R \cup R' = R'$ y las variables en Γ^P con tipo r son exactamente las contenidas en R'.

- $x \in S'$

  Sea $\Gamma^P = [x : s] + [y : r \mid y \in R'] + [r : \rho]$. Este entorno está bien definido por el Lema 3 (7). La regla de $\vdash_{check}$ utilizada es [COPY3$_C$]. Vuelve a cumplirse $\Gamma^P \geq_{e'} [x : s, r : \rho]$, ya que $\Gamma^P(x) = s \geq s$. Por tanto, $\Gamma^P \vdash e' : s$.

  Además también se verifica (2), ya que $D \cup D' = \emptyset$, $R \cup R' = R'$ y $S \cup S' = \{x\}$.

$\boxed{e' = C\ \overline{a_i}^n\ @r}$

$$\frac{\Sigma(C) = \sigma \quad \overline{s_i}^n \to \rho \to T\ @\rho \trianglelefteq \sigma \quad \Gamma^P = \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma^P \vdash (C\ \overline{a_i}^n)@r : T\ @\rho}\ [\text{CONS}]$$

$$\frac{\forall i \in \{1..n\}.a_i \vdash_{inf} (\emptyset, \emptyset, S_i, \emptyset)}{C\overline{a_i}^n@r \vdash_{inf} (\emptyset, \emptyset, \bigcup_{i=1}^n S_i, \emptyset)}\ [\text{CONS}_I] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} C\ \overline{a_i}^n\ @r}\ [\text{CONS}_C]$$

Sea $\Gamma^P = \bigoplus_{i=1}^n [a_i : s] + [y : r \mid y \in R'] + [r : \rho]$. El resultado de $\bigoplus$ está bien definido, ya que sólo se realiza sobre entornos con variables seguras. Además $S =$

$\{a_i \mid var(a_i) \wedge i \in \{1..n\}\}$ y según el Lema 3 (7) se cumple $R' \cap S = \emptyset$. Por consiguiente, Γ^P está bien definido.

Por otra parte, sea $\Gamma'^P = \bigoplus_{i=1}^n [a_i : s] + [r : \rho]$. Trivialmente se cumple $\Gamma'^P \vdash e' : s$. A partir de esto y mediante el uso reiterado de la regla [EXTS] puede ampliarse el entorno con elementos de R' hasta obtener $\Gamma^P \vdash e' : s$.

Además $R \cup R' = R'$ y $D \cup D' = \emptyset$, por lo que se cumplen las propiedades (2a) y (2c). Por otro lado se tiene para toda variable x:

$$\Gamma^P(x) = s \Leftrightarrow x \in var(e') \Leftrightarrow x \in S \cup S'$$

Por lo que la propiedad (2b) también es cierta.

$\boxed{e' = g\ \overline{a_i}^n\ @r}$

$$\frac{\overline{t_i}^n \to \rho \to T\ @\rho \trianglelefteq \sigma \qquad \Gamma^P = [g : \sigma] + [r : \rho] + \bigoplus_{i=1}^n [a_i : t_i] \quad R = \bigcup_{i=1}^n \{sharerec(a_i, (f\ \overline{a_i}^n)@r) - \{a_i\} \mid cdm?(t_i)\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\}}{\Gamma_R + \Gamma^P \vdash g\ \overline{a_i}^n\ @r : T\ @\rho} \text{ [APP1]}$$

$$\frac{\begin{array}{ccc} \forall i \in \{1..n\}.D_i = \{a_i \mid i \in I_D\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n S_i) = \emptyset & R \cap (\bigcup_{i=1}^n S_i) = \emptyset \\ \forall i \in \{1..n\}.S_i = \{a_i \mid i \in I_S\} & (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n N_i) = \emptyset & R \cap (\bigcup_{i=1}^n D_i) = \emptyset \\ \forall i \in \{1..n\}.N_i = \{a_i \mid i \in I_N\} & \forall i, j \in \{1..n\}\ .\ i \neq j \Rightarrow D_i \cap D_j = \emptyset & R \cap (\bigcup_{i=1}^n N_i) = \emptyset \\ \Sigma \vdash f : (I_D, \emptyset, I_S, I_N) & R = \bigcup_{i=1}^n \{sharerec(a_i, g\overline{a_i}^n@r) - \{a_i\} \mid a_i \in D_i\} & \end{array}}{g\ \overline{a_i}^n@r \vdash_{inf} (\bigcup_{i=1}^n D_i, R, \bigcup_{i=1}^n S_i, (\bigcup_{i=1}^n N_i) - (\bigcup_{i=1}^n S_i))} \text{ [APP}_I\text{]}$$

$$\frac{g\overline{a_i}^n@r \vdash_{inf} (D, R, S, N) \qquad \forall a_i \in D_p\ .\ (\#j : 1 \leq j \leq n : a_i = a_j) = 1}{(D_p, R_p, S_p) \vdash_{check} g\ \overline{a_i}^n@r} \text{ [APP}_C\text{]}$$

Suponemos que f es una llamada no recursiva, es decir, $f \neq g$. En este caso se cumple $N = \emptyset$ y por tanto, $D' = S' = \emptyset$.

Sea $(I_D, \emptyset, I_S, \emptyset)$ la signatura obtenida para g. Para cada $i \in \{1..n\}$ se define Γ_i^P del siguiente modo:

$$\Gamma_i^P = \begin{cases} [a_i : d] & \text{si } a_i \text{ es variable e } i \in I_D \\ [a_i : s] & \text{si } a_i \text{ es variable e } i \in I_S \end{cases} \qquad (i > 0)$$

Se define Γ'^P de la siguiente forma:

$$\Gamma'^P = [r : \rho, g : \sigma] + \bigoplus_{i=0}^{n} \Gamma_i^P$$

Demostramos por reducción al absurdo que Γ'^P está bien definido. Supongamos que no está definido. Puede deberse a tres causas:

1. Existen dos entornos Γ_i^P y Γ_j^P, (con $i, j \in \{1..n\}$, $i \neq j$) tales que $x \in dom(\Gamma_i^P) \cap dom(\Gamma_j^P)$ y $\Gamma_i^P(x) = \Gamma_j^P(x) = d$. Entonces se tiene $i, j \in I_D$ y por hipótesis de la regla [APP$_I$], $x \in D_i$ y $x \in D_j$. Esto lleva a obtener $D_i \cap D_j \neq \emptyset$, lo cual contradice una de las premisas de la regla [APP$_I$].

2. Existen dos entornos Γ_i^P y Γ_j^P, (con $i, j \in \{1..n\}$, $i \neq j$) tales que $x \in dom(\Gamma_i^P) \cap dom(\Gamma_j^P)$ y $\Gamma_i^P(x) = d$ y $\Gamma_j^P(x) = s$. En este caso $x \in D_i$ y $x \in S_j$, lo cual contradice la premisa $(\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n S_i) = \emptyset$.
3. Existen dos entornos Γ_i^P y Γ_j^P, (con $i, j \in \{1..n\}$, $i \neq j$) tales que $x \in dom(\Gamma_i^P) \cap dom(\Gamma_j^P)$ y $\Gamma_i^P(x) = s$ y $\Gamma_j^P(x) = d$. Este caso es análogo al anterior.

Por otro lado se construyen dos entornos Γ_R y $\Gamma_{R'}$ de la siguiente forma:

$$\begin{array}{rcl} \Gamma_R & = & [y : r \mid y \in sharerec(a_i, e') - \{a_i\}, i \in \{1..n\}, cdm?(t_i)] \\ \Gamma_{R'} & = & [y : r \mid y \in R'] \end{array}$$

y se define $\Gamma''^P = \Gamma_R + \Gamma'^P$. Puede verse que no hay variables comunes en los dominios de Γ_R y Γ'^P y que por tanto, Γ''^P está bien definido. En efecto, la propia definición Γ_R excluye a los a_i y a los identificadores g y r que conforman el dominio de Γ'^P.

Por el modo en el que se ha construido el entorno Γ''^P, análogo a la regla de tipos [APP], puede concluirse que se verifica $\Gamma''^P \vdash e' : s$. Definimos ahora $\Gamma^P = \Gamma_{R'} \otimes \Gamma''^P$. El Lema 1 (7) garantiza que Γ^P está bien definido. Partiendo de $\Gamma''^P \vdash e' : s$ y mediante la regla [EXTS] puede obtenerse $\Gamma^P \vdash e' : s$, por lo que (1) se verifica. Igualmente resulta sencillo comprobar que la propiedad (2) es cierta:

$$\begin{array}{rcl} \Gamma^P(x) = d & \Leftrightarrow & \exists i \in \{1..n\}.\ x \in dom(\Gamma_i^P) \ \wedge \ \Gamma_i^P(x) = d \\ & \Leftrightarrow & \exists i \in I_D.\ x = a_i \\ & \Leftrightarrow & \exists i \in \{1..n\}\ .\ x \in D_i \\ & \Leftrightarrow & x \in D \\ \Gamma^P(x) = s & \Leftrightarrow & \exists i \in \{1..n\}.\ x \in dom(\Gamma_i^P) \ \wedge \ \Gamma_i^P(x) = s \\ & \Leftrightarrow & \exists i \in I_S.\ x = a_i \\ & \Leftrightarrow & \exists i \in \{1..n\}\ .\ x \in S_i \\ & \Leftrightarrow & x \in S \\ \Gamma^P(x) = r & \Leftrightarrow & x \in dom(\Gamma_R) \ \vee \ x \in dom(\Gamma_{R'}) \\ & \Leftrightarrow & (\exists i \in \{1..n\}.\ x \in sharerec(a_i, e') - \{a_i\} \ \wedge \ a_i \in D_i) \ \vee \ x \in R' \\ & \Leftrightarrow & x \in R \cup R' \end{array}$$

$\boxed{e' = \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}$

Suponemos el caso $x_1 \notin D_2 \cup R_2$, correspondiente al uso de la regla [LET1] del sistema de tipos.

$$\frac{\Gamma_1^P \vdash e_1 : s_1 \quad \Gamma_2^P + [x_1 : s_1] \vdash e_2 : s \quad C = shareall(x_1, e_2)}{\Gamma_1^P \rhd_C^{fv(e)} \Gamma_2^P \vdash \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 : s}\ [\text{LET1}]$$

$$\frac{\begin{array}{c} C = \begin{cases} shareall(x_1, e_2) & \text{si } x_1 \notin D_2 \cup R_2 \\ shareall(x_1, e_2) - sharerec(x_1, e_2) & \text{si } x_1 \in D_2 \end{cases} \\ e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \qquad N = (N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2 \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \qquad def((D_1 \cup R_1) \rhd_C^{FV(e_2)} (D_2 \cup R_2)) \\ (\emptyset, \emptyset, N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{check} e_1 \qquad (\emptyset, \emptyset, (S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2 \end{array}}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash_{inf} ((D_1 \cup D_2) - \{x_1\}, R_1 \cup (R_2 - D_1), ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2), N - \{x_1\})}\ [\text{LET}_I]$$

$$\frac{\begin{array}{c} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \quad (D_p \cap N_1, R_p - D_1, S_p \cap N_1) \vdash_{check} e_1 \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad (D_p \cap N_2, (R_p \cup R'_p) - D_2, S_p \cap N_2) \vdash_{check} e_2 \\ R'_p = \{y \in ((D_p \cap N_1) \cup D_1) \cap sharerec(z, e_2) \mid z \in D_p \cap N_2\} - (N_2 \cup \{x_1\}) \\ R_p \cap (S_1 \cup S_2) = \emptyset \qquad x_1 \notin S_2 \cup N_2 \ \vee \ \forall z \in D_p \cap N_2 \,.\, x_1 \notin sharerec(z, e_2) \\ L = FV(e_2) \qquad C = \begin{cases} shareall(x_1, e_2) & \text{si } x_1 \in S_2 \\ shareall(x_1, e_2) - sharerec(x_1, e_2) & \text{si } x_1 \in D_2 \end{cases} \\ R''_p = \{sharerec(z, e_1) \mid z \in D_p \cap N_1\} \\ def((D_p \cap N_1) \cup (R_p - D_2) \cup R''_p \rhd^L_C (D_p \cap N_2) \cup ((R_p \cup R'_p) - D_2)) \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{let } x_1 = e_1 \textbf{ in } e_2} \ [\text{LET}_C]$$

Se tiene:

$$\begin{array}{ll} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) & e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \\ (D'_1, R'_1, S'_1) \vdash^*_{check} e_1 & (D'_2, R'_2, S'_2) \vdash^*_{check} e_2 \end{array}$$

Por hipótesis de inducción existen dos entornos Γ_1^P y Γ_2^P que cumplen:

$$\Gamma_1^P \vdash e_1 : s_1 \qquad \Gamma_2^P \vdash e_2 : s_2$$

Se define $\Gamma_2'^P = \Gamma_2^P \upharpoonright (dom(\Gamma_2^P) - \{x_1\})$ y $\Gamma^P = \Gamma_1^P \rhd^L_C \Gamma_2'^P$, donde $L = FV(e_2)$ y $C = shareall(x_1, e_2)$. El entorno Γ^P estará bien definido si se cumplen las dos condiciones que se demuestran a continuación:

1. $\forall x \in dom(\Gamma_1^P).\, unsafe?(\Gamma_1^P(x)) \Rightarrow x \notin L$

   Sea x una variable del dominio de Γ_1^P tal que $x \in L$. El uso del operador $\rhd$ en la regla [LET$_I$] prohíbe que x pertenezca a $D_1 \cup R_1$. Por otro lado, el uso del operador $\rhd$ en la regla [LET$_C$] prohíbe que x pertenezca a $D'_1 \cup R'_1$.

   A partir de $x \notin D_1 \cup D'_1$ se obtiene por hipótesis de inducción $\Gamma_1^P(x) \neq d$.

   A partir de $x \notin R_1 \cup R'_1$ se obtiene por hipótesis de inducción $\Gamma_1^P(x) \neq r$.

   Reuniendo estos dos resultados se deduce $\neg unsafe?(\Gamma_1^P(x))$.

2. $\forall x \in C \cap L \cap dom(\Gamma_2'^P).\, \neg unsafe?(\Gamma_2'^P(x))$

   Sea x una variable del dominio de $\Gamma_2'^P$ tal que $x \in L$ y $x \in C$. El operador $\rhd$ de la regla [LET$_I$] impide que $x \in D_2 \cup R_2$. El operador $\rhd$ de la regla [LET$_C$] impide que $x \in D'_2 \cup R'_2$. Por tanto, se cumple $\Gamma_2'^P(x) \neq d, r$, que implica $\neg unsafe?(\Gamma_2'^P(x))$.

Con ello se demuestra que Γ^P está bien definido. Ahora es necesario demostrar que $\Gamma_2'^P + [x_1 : s_1] \vdash e_2 : s$. Si $x_1 \notin dom(\Gamma_2^P)$ entonces $\Gamma_2'^P = \Gamma_2^P$. En este caso se tiene $\Gamma_2'^P \vdash e_2 : s$ y basta con utilizar la regla [EXTS] para obtener $\Gamma_2'^P + [x_1 : s_1] \vdash e_2 : s$. Si $x_1 \in dom(\Gamma_2^P)$ hemos de demostrar que $\Gamma_2^P(x_1) = s$. Lo haremos por reducción al absurdo: Supongamos $\Gamma_2^P(x_1) = d$. Entonces $x_1 \in D_2 \cup D'_2$. Como estamos considerando el caso $x_1 \notin D_2 \cup R_2$ (regla [LET1] del sistema de tipos), ha de cumplirse $x_1 \in D'_2$. Esto implica $x_1 \in N_2$ (Lema 3 (5)). Por tanto, el $\vdash_{check}$ sobre e_2 contenido en la regla [LET$_I$] fuerza a que $x_1 \in S'_2$. Esto contradice el hecho la propiedad $D'_2 \cap S'_2 = \emptyset$ (Lema 3 (4)). Mediante un razonamiento similar puede demostrarse que $\Gamma_2^P(x_1) \neq r$. La única posibilidad es $\Gamma_2^P(x_1) = s$.

Hemos obtenido $\Gamma^P \vdash e' : s$, por lo que la propiedad (1) es cierta. Pasamos a demostrar la propiedad (2a) considerando cada implicación por separado. Sea $x \in \mathit{scope}(e')$. Esto implica que $x \neq x_1$:

- $x \in D \cup D' \Rightarrow \Gamma^P(x) = d$

  Si $x \in D$, entonces por definición de D se obtiene $x \in D_1 \ \vee \ x \in D_2$.

  Si $x \in D'$, entonces $x \in N$. Por la definición de N puede obtenerse $x \in N_1 \ \vee \ x \in N_2$ y por tanto, $x \in D'_1 \ \vee \ x \in D'_2$.

  En el caso en que $x \in D_1 \cup D'_1$ se obtiene $\Gamma^P_1(x) = d$ por hipótesis de inducción y por tanto, $\Gamma^P(x) = (\Gamma^P_1 \rhd^L_C \Gamma^P_2)(x) = \Gamma^P_1(x) = d$

  Por otro lado, en el caso en que $x \in D_2 \cap D'_2$ se consigue $\Gamma^P_2(x) = d$ por hipótesis de inducción y se cumple $\Gamma^P(x) = (\Gamma^P_1 \rhd^L_C \Gamma^P_2)(x) = \Gamma^P_2(x) = d$.

- $\Gamma^P(x) = d \Rightarrow x \in D \cup D'$

  Si $\Gamma^P(x) = \Gamma^P_1(x) = d$ entonces $x \in D_1 \cup D'_1$ y por tanto, $x \in D \cup D'$.

  Si $\Gamma^P(x) = \Gamma^P_2(x) = d$ entonces $x \in D_2 \cup D'_2$ y por tanto, $x \in D \cup D'$.

Con ello queda demostrada la propiedad (2a). Pasamos a demostrar (2b):

- $x \in S \cup S' \Rightarrow \Gamma^P(x) = s$

  Suponemos $x \in S$. Sabemos que $x \in (S_1 - N_2) \cup S_2$ y que $x \notin D_2 \cup R_2$.

  Si $x \in S_1$ pero $x \notin N_2$ por hipótesis de inducción se tiene $\Gamma^P_1(x) = s$. Para poder deducir $(\Gamma^P_1 \rhd^L_C \Gamma^P_2)(x) = \Gamma^P_1(x) = s$ basta con demostrar que x no puede aparecer con tipo inseguro en Γ^P_2. En efecto:

  - Si $\Gamma^P_2(x) = d$ entonces $x \in D_2 \cup D'_2$, lo cual no es posible, ya que $x \notin D_2 \cup N_2$.
  - Si $\Gamma^P_2(x) = r$ se tiene $x \in R_2 \cup R'_2$. Como $x \notin R_2$ entonces se ha de cumplir $x \in R'_2$. La aparición de la variable x en R'_2 sólo puede provenir de una llamada a $\vdash_{\mathit{check}}$ sobre e_2 en la regla [LET$_C$]. Por tanto, se tiene $x \in R' \cup R'_p$. No es posible que $x \in R'$, ya que $R' \cap S = \emptyset$ por el Lema 3 (7). Tampoco es posible que $x \in R'_p$, ya que esto implicaría $x \in N_1 \cup D_1$, lo cual contradice el Lema 3 (1).

  Si $x \in S_2$ se tiene $\Gamma^P_2(x) = s$ por hipótesis de inducción. Esto implica que $x \in FV(e_2)$. Como el entorno Γ^P está bien definido no puede ocurrir bajo estas condiciones que x tenga tipo inseguro en Γ^P_1. Por tanto, $\Gamma^P(x) = (\Gamma^P_1 \rhd^L_C \Gamma^P_2) = s$.

  Ahora suponemos $x \in S'$. Por el Lema 3 (5) se tiene $x \in N \subseteq N_1 \cup N_2$.

  Si $x \in N_1$ entonces $x \in S'_1$. Por hipótesis de inducción se tiene $\Gamma^P_1(x) = s$. Para demostrar $\Gamma^P(x) = (\Gamma^P_1 \rhd^L_C \Gamma^P_2)(x) = s$ es necesario comprobar que x no tiene tipo inseguro en Γ^P_2:

- Si $\Gamma_2^P(x) = d$ entonces $x \in D_2 \cup D_2'$. Sin embargo, no es posible que $x \in D_2$ ya que implicaría $x \in D$, lo cual contradice el hecho de que D y N son disjuntos. Tampoco es posible $x \in D_2'$, ya que conlleva tener $x \in D'$, que contradice el hecho de que D' y S' son disjuntos.
- Si $\Gamma_2^P(x) = r$ entonces $x \in R_2 \cup R_2'$. No es posible que $x \in R_2$ porque ello implica $x \in R \cup D$ y los conjuntos D y R son disjuntos con N. Tampoco se puede dar el caso $x \in R_2'$. Esto implicaría $x \in R' \cup R_p'$. Si $x \in R'$ entonces R' y S' no serían disjuntos. Si $x \in R_p'$ se tiene $x \in D' \cup D_1 \subseteq D' \cup D$. Sin embargo $D' \cap S' = \emptyset$ y $D \cap N = \emptyset$ por lo que hemos llegado a una contradicción.

Si $x \in N_2$ entonces $x \in S_2'$ y, por hipótesis de inducción, $\Gamma_2^P(x) = s$. Como $x \in FV(e_2)$ y Γ^P está bien definido la variable x no puede aparecer con tipo inseguro en Γ_1^P. Por tanto, $\Gamma^P(x) = (\Gamma_1^P \rhd_C^L \Gamma_2^P)(x) = s$.

- $\Gamma^P(x) = s \Rightarrow x \in S \cup S'$

  Si $\Gamma^P(x) = \Gamma_1^P(x) = s$ entonces $x \in S_1 \cup S_1'$. También puede deducirse $x \notin D_2$ y $x \notin R_2$, ya que $\Gamma_2^P(x) \neq d, r$. En el caso $x \in S_1'$ se obtiene $x \in S'$. Por otra parte, si $x \in S_1$ distinguimos casos:

  - Si $x \notin N_2$ se cumple $x \in S$.
  - Si $x \in N_2$, entonces $x \in N \subseteq D' \cup R' \cup S'$. No es posible que $x \in D'$, ya que implica $\Gamma^P(x) = d$. Tampoco es posible $x \in R'$ ya que, como se verá, implica $\Gamma^P(x) = r$. Por tanto, $x \in S'$. (llamada a $\vdash_{check}$ en [LET$_I$]).

  Si $\Gamma^P(x) = \Gamma_2^P(x) = s$ entonces $x \in S_2 \cup S_2'$. Además se tiene $x \notin D_2$ y $x \notin R_2$, ya que $\Gamma_2^P(x) \neq d, r$. Bajo estas condiciones, si $x \in S_2$ entonces $x \in S$. Además, si S_2' entonces $x \in S'$.

Por último se demostrará (2c):

- $x \in R \cup R' \Rightarrow \Gamma^P(x) = r$

  Sea $x \in R \subseteq R_1 \cup R_2$. Si $x \in R_1$ entonces $\Gamma_1^P(x) = r$ y por tanto, $\Gamma^P(x) = (\Gamma_1^P \rhd_C^L \Gamma_2^P)(x) = r$. Si $x \in R_2$ entonces $\Gamma_2^P(x) = r$. En este caso se cumplirá $\Gamma^P(x) = r$ sólo si x no tiene tipo d en Γ_1^P. Sin embargo, esto último no puede ocurrir, ya que Γ^P está bien definido.

  Por otra parte, sea $x \in R'$. Esto fuerza a que $x \notin D_1$, ya que en caso contrario se tendría $x \in D$, lo que contradice $R' \cap D = \emptyset$ (Lema 3 (7). Por tanto, se tiene $x \in R_1'$ y por hipótesis de inducción, $\Gamma_1^P(x) = r$. Por consiguiente, $\Gamma^P(x) = r$.

- $\Gamma^P(x) = r \Rightarrow x \in R \cup R'$

  Si $\Gamma^P(x) = \Gamma_1^P(x) = r$ entonces $x \in R_1 \cup R_1'$ y por tanto, $x \in R \cup R'$.

  Si $\Gamma^P(x) = \Gamma_2^P(x) = r$ entonces ha de cumplirse $\Gamma_1^P(x) \neq d$, lo cual implica $x \notin D_1$. Por otro lado la hipótesis de inducción establece que $x \in R_2 \cup R_2'$. Si

$x \in R'_2$ entonces $x \in R' \cup R'_p$. No es posible que $x \in R'_p$, ya que implica $x \in D' \cup D_1 \subseteq D' \cup D$, lo cual implicaría $\Gamma^P(x) = d$. Por tanto, $x \in R'$.

En el caso $x \in D_2$ (regla [LET2] del sistema de tipos) el razonamiento es similar. La única diferencia es que es necesario demostrar en este caso la premisa $\Gamma'^P_2 + [x_1 : d_1] \vdash e_2 : s$. Sin embargo, al tener $x \in D_2$ se tiene siempre que $x \in dom(\Gamma^P_2)$ y además $\Gamma^P_2(x) = d$, por lo que la premisa es cierta.

$$\boxed{e' = \textbf{case } z \textbf{ of } \overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

$$\frac{\begin{array}{c}(\forall i \in \{1..n\}).\Sigma(C_i) = \sigma_i \qquad (\forall i \in \{1..n\}).\overline{s_{ij}}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \\ \Gamma^P \geq_{\textbf{case } z \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}} [z : T@\rho] \qquad (\forall i \in \{1..n\}.\forall j \in \{1..n_i\}).inh(\tau_{ij}, s_{ij}, \Gamma^P(z)) \\ (\forall i \in \{1..n\}).\Gamma^P + \overline{[x_{ij} : \tau_{ij}]}^{n_i} \vdash e_i : s\end{array}}{\Gamma^P \vdash \textbf{case } z \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} : s}\ [\text{CASE}]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}\,.\,e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\}\,.\,P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad (D, R, S, N) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \\ \forall i \in \{1..n\}\,.\,Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \quad \forall i \in \{1..n\}\,.\,def(inh(type(z), D_i, R_i, S_i, P_i, Rec_i)) \\ type(z) = \begin{cases} d & \text{si } z \in D \\ r & \text{si } z \in R \\ s & \text{si } z \in S \\ n & \text{e. o. c.} \end{cases} \quad N' = \begin{cases} N & \text{si } z \in D \cup R \cup S \\ N \cup \{z\} & \text{si } z \notin D \cup R \cup S \end{cases} \\ \forall i \in \{1..n\}\,.\,R''_i = \{y \in P_i \cap sharerec(z, e_i) \mid z \in (D \cup D'_i) \cap N_i\} \\ \forall i \in \{1..n\}\,.\,R'''_i = \{y \in D \cap sharerec(z, e_i) \mid z \in (D \cup D'_i) \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\,R''_i \cap (S_i \cup S'_i) = \emptyset \\ \forall i \in \{1..n\}\,.\,((D \cup D'_i) \cap N_i, R \cup ((R'_i \cup R''_i \cup R'''_i) - D_i), (S \cup S'_i) \cap N_i) \vdash_{check} e_i \\ \textbf{donde } D'_i = \emptyset \quad R'_i = \begin{cases} Rec_i & \text{si } type(z) = d \\ \emptyset & \text{e.o.c.} \end{cases} \quad S'_i = \begin{cases} P_i & \text{si } type(z) = s \\ P_i - (R_i \cup S_i) & \text{si } type(z) = r \\ P_i - Rec_i & \text{si } type(z) = d \\ \emptyset & \text{e.o.c.} \end{cases}\end{array}}{\textbf{case } z \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \vdash_{inf} (D, R, S, N')}\ [\text{CASE}_I]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}\,.\,e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad \forall i \in \{1..n\}\,.\,D_{p_i} = \emptyset \\ \forall i \in \{1..n\}\,.\,P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad \forall i \in \{1..n\}\,.\,R_{p_i} = \begin{cases} Rec_i & \text{si } z \in D_p \\ \emptyset & \text{e.o.c.} \end{cases} \\ \forall i \in \{1..n\}\,.\,Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} | j \in RecPos(C_i)\} \\ type(z) = \begin{cases} d & \text{si } z \in D_p \\ r & \text{si } z \in R_p \\ s & \text{si } z \in S_p \\ n & \text{e.o.c.} \end{cases} \qquad \forall i \in \{1..n\}\,.\,S_{p_i} = \begin{cases} P_i - Rec_i & \text{si } z \in D_p \\ P_i & \text{si } z \in S_p \\ P_i - (R_i \cup S_i) & \text{si } z \in R_p \\ \emptyset & \text{e.o.c.} \end{cases} \\ \forall i \in \{1..n\}\,.\,R'_{p_i} = \{x \in P_i \cap sharerec(z, e_i) \mid z \in (D_p \cup D_{p_i}) \cap N_i\} \\ \forall i \in \{1..n\}\,.\,R''_{p_i} = \{y \in ((D_p \cap N) \cup D) \cap sharerec(z, e_i) \mid z \in (D_p \cup D_{p_i}) \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\}\,.\,R'_{p_i} \cap (S_i \cup S_{p_i}) = \emptyset \ \wedge\ R_p \cap S_i = \emptyset \\ \forall i \in \{1..n\}\,.\,((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \\ z \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\}\,.\,def(inh(type(z), D_i, R_i, S_i, P_i, Rec_i))\end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case } z \textbf{ of } \overline{C_i \overline{x_{ij}}^{n_i} \to e_i}^{n}}\ [\text{CASE}_C]$$

Definimos:

$$\begin{array}{rcl}\Gamma^P & = & [x : d \mid x \in D \cup D'] \\ & + & [x : r \mid x \in R \cup R'] \\ & + & [x : s \mid x \in S \cup S']\end{array}$$

En primer lugar demostramos que Γ^P está bien definido. Para ello mostraremos que los conjuntos $(D \cup D')$, $(R \cup R')$ y $(S \cup S')$ son disjuntos dos a dos. Demostramos $(D \cup D') \cap (R \cup R') = \emptyset$. En efecto, mediante el Lema 3 (1) se obtiene $D \cup R = \emptyset$. Por otro lado, como $D' \subseteq N$ y $N \cap R = \emptyset$, se tiene $D' \cap R = \emptyset$. Las igualdades $D \cap R' = \emptyset$ y $D' \cap R' = \emptyset$ se obtienen aplicando las propiedades (7) y (4), respectivamente, del Lema 3. La igualdad $(R' \cup R) \cap (S \cup S') = \emptyset$ se demuestra de modo similar. Con respecto a $(D \cup D') \cap (S \cup S') = \emptyset$ puede utilizarse el Lema 3 (1, 6).

A continuación se mostrará $\Gamma^P \geq_{e'} [z : s]$. En primer lugar $z \in dom(\Gamma^P)$, ya que al ser z una variable libre en e', por el Lema 3 (5) tenemos $z \in D$, o bien $z \in R$, o bien $z \in S$, o bien $z \in N \subseteq D' \cup R' \cup S'$. Además, en cualquier caso se cumple $\Gamma^P(z) \geq s$. Por último, reuniendo los resultados de las propiedades (3) y (6) del Lema 3 se tiene:

$$\bigcup_{z \in D \cup D'} sharerec(z, e') \subseteq D \cup R \cup D' \cup R'$$

Por lo que toda variable que comparta un hijo recursivo de otra con tipo d en Γ^P tendrá tipo inseguro en este entorno. Con esto se verifican las tres condiciones para que $\Gamma^P \geq_{e'} [z : s]$.

A continuación demostraremos que $\Gamma_i'^P \vdash e_i : s$, donde $\Gamma_i'^P = \Gamma^P + \overline{[x_{ij} : \tau_{ij}]}^{n_i}$ y se hallarán los τ_{ij} correspondientes para que se verifiquen los predicados *inh* de la regla [CASE]. En primer lugar sabemos que existe un Γ_i^P que verifica $\Gamma_i^P \vdash e_i : s$, por hipótesis de inducción. Si demostramos que $\forall x \in dom(\Gamma_i^P).\, \Gamma_i^P(x) = \Gamma_i'^P(x)$ podrá deducirse, mediante aplicaciones de las reglas [EXTS] y [EXTD] que $\Gamma_i'^P \vdash e' : s$.

Sea $x \in dom(\Gamma_i^P) \cap dom(\Gamma_i'^P)$ y supongamos que $x \in P_i$, es decir, $x = x_{ij}$ para algún $j = \{1..n_i\}$. Distinguimos casos según el tipo SAFE del discriminante del **case**:

- $z \in D$

  Supongamos $x \in P_i - Rec_i$. El uso de *inh* en la regla [CASE$_I$] impone que $x \notin D_i$ y $x \notin R_i$. Tenemos que, o bien $x \in S_i$, o bien $x \in N_i$. En este último caso sólo cabe la posibilidad de que $x \in S_i'$ ya que $D_i' = \emptyset$ y R_i' sólo contiene elementos de Rec_i (en caso contrario, la relación *inh* en [CASE$_C$] no estaría definida). Dado que $x \in S_i \cup S_i'$, tenemos que $\Gamma_i^P(x) = s$ por hipótesis de inducción. Esto nos obliga a asignar $\tau_{ij} = s$ dentro del entorno $\Gamma_i'^P$. De este modo se cumplirá el predicado $inh(s, s, \Gamma^P(z))$ de la regla [CASE] del sistema de tipos, ya que $\Gamma^P(z)$ tiene tipo d y se cumple $\neg utype?(\Gamma'^P(x), \Gamma^P(z))$, ya que $x \notin Rec_i$.

  Ahora supongamos $x \in Rec_i$. El mismo *inh* de la regla [CASE$_I$] especifica que $x \notin D_i$ y $x \notin S_i$. La única posibilidad es que $x \in R_i$ o que $x \in N_i$. En este último caso ha de cumplirse $x \in R_i'$, ya que D_i' es vacío y S_i' no contiene elementos de Rec_i. Dado que $x \in R_i \cup R_i'$ y por tanto, $\Gamma_i^P(x) = r$, podemos asignar $\tau_{ij} = r$ para

que $\Gamma'^P(x) = r$. Con esto se cumplirá del mismo modo el predicado *inh* de la regla [CASE] del sistema de tipos para esta variable x.

- $z \in R$

  El predicado *inh* de [CASE$_I$] obliga a que $x \notin D_i$. Por otro lado tenemos $D'_i = \emptyset$ y $S'_i = P_i - (R_i \cup S_i)$. Las únicas posibilidades son $x \in R_i \cup R'_i$ y $x \in S_i \cup S'_i$.

  Si $x \in R_i \cup R'_i$ se tiene $\Gamma_i^P(x) = r$ y asignando $\tau_{ij} = r$ para que $\Gamma_i'^P(x) = r$ permite que se verifique el predicado *inh* de la regla [CASE].

  Si $x \in S_i \cup S'_i$ entonces $\Gamma_i^P(x) = s$ por lo que puede asignarse $\tau_{ij} = s$ y el predicado *inh* de la regla [CASE] se verifica también para x.

- $z \in S$

  En este caso se tiene $x \notin D_i$ y $x \notin R_i$, y si $x \in N_i$ entonces $x \in S'_i$. El caso $x \in R'_i$ no podría darse, ya que implica $x \in R''_i$ y provocaría que z compartiese un hijo recursivo de algún $D \cup D'_i$, lo cual contradice el hecho de que $z \in S$. Por tanto se tiene $x \in S_i \cup S'_i$ y por hipótesis de inducción, $\Gamma_i^P(x) = s$. Puede asignarse $\tau_{ij} = s$ y se verificará el predicado *inh* de la regla [CASE].

- $z \in N$

  Puede realizarse distinción de casos según se cumpla $z \in D'$, $z \in R'$ o $z \in S'$. La demostración para cada caso es similar a las tres anteriores. Esta vez son el *inh* de la regla [CASE$_C$] y las definiciones de D_{p_i}, R_{p_i} y S_{p_i} los que fuerzan a x tener un tipo determinado. Este tipo permitirá que se verifiquen los predicados *inh* correspondientes en el sistema de tipos.

Ahora suponemos que $x \notin P_i$. Si $x \in D_i$ entonces $\Gamma_i^P(x) = d$ por hipótesis de inducción. En ese caso con la definición de $\sqcup$ se obtiene $x \in D$ y por tanto, $\Gamma_i'^P(x) = \Gamma^P(x) = d$. Para los casos $x \in R_i$ y $x \in S_i$ se realiza un razonamiento similar. Si $x \in N_i$ entonces se distinguen casos según se verifique $x \in D'_i$, $x \in S'_i$, $x \in R'_i$. En el primer caso se tiene $\Gamma_i^P(x) = d$ y dado que $x \in D'_i \cap N_i$ entonces se tiene $x \in D'$, por lo que $\Gamma_i'^P(x) = \Gamma_i^P(x) = d$. De nuevo, en los dos casos restantes se realiza un razonamiento análogo.

Con lo visto hasta el momento puede derivarse $\Gamma^P \vdash e' : s$ mediante la regla [CASE] del sistema de tipos y por tanto, se cumple la propiedad (1). Por su parte, las propiedades (2a), (2b) y (2c) se cumplen trivialmente por la definición de Γ^P.

$$\boxed{e' = \mathbf{case!}\ z\ \mathbf{of}\ \overline{C\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

$$\frac{\begin{array}{ll}(\forall i \in \{1..n\}).\ \Sigma(C_i) = \sigma_i & (\forall i \in \{1..n\}).\ \overline{s_{ij}}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \\ R = \mathit{sharerec}(z, \mathbf{case!}\ z\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}) - \{z\} & (\forall i \in \{1..n\}.\ \forall j \in \{1..n_i\}).\mathit{inh}!(t_{ij}, s_{ij}, T\ !@\rho) \\ \forall z \in R \cup \{z\}, i \in \{1..n\}.z \notin \mathit{fv}(e_i) & (\forall i \in \{1..n\}).\ \Gamma^P + [z : T\ \#@\rho] + \overline{[x_{ij} : t_{ij}]}^{n_i} \vdash e_i : s \\ \multicolumn{2}{c}{\Gamma_R = \{y : \mathit{danger}(\mathit{type}(y)) \mid y \in R\}}\end{array}}{\Gamma_R \otimes \Gamma^P + [z : T\ !@\rho] \vdash \mathbf{case!}\ z\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} : s}\ [\text{CASE!}]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\} \,.\, P_i = \bigcup_{j=1}^{n_i}\{x_{ij}\} \qquad (D, R', S, N) = \sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \\ \forall i \in \{1..n\} \,.\, Rec_i = \bigcup_{j=1}^{n_i}\{x_{ij} \mid j \in RecPos(C_i)\} \quad \forall i \in \{1..n\} \,.\, def(inh!(D_i, R_i, S_i, P_i, Rec_i)) \\ R = sharerec(z, \textbf{case!}\ z\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}) - \{z\} \qquad def((R \cup \{z\}) \rhd^{L}_{\emptyset} (D \cup R')) \\ L = \bigcup_{i=1}^{n} FV(e_i) \qquad type(z) = T@\rho \\ \forall i \in \{1..n\} \,.\, ((D \cup Rec_i) \cap N_i, R' \cup ((R'_i \cup R''_i \cup R'''_i) - D_i), (S \cup (P_i - Rec_i)) \cap N_i) \vdash_{check} e_i \\ \textbf{donde}\ R'_i = \{y \in P_i \cap sharerec(z, e_i) \mid z \in D \cap N_i\} - ((D \cup Rec_i) \cap N_i) \\ R''_i = \{y \in sharerec(x, e_i) \mid\ x \in Rec_i \cap N_i\} - ((D \cup Rec_i) \cap N_i) \\ R'''_i = \{y \in D \cap sharerec(z, e_i) \mid z \in D \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\} \,.\, R'_i \cap S_i = \emptyset \ \wedge\ R'_i \cap (P_i - Rec_i) = \emptyset \ \wedge\ R''_i \cap S = \emptyset \end{array}}{\textbf{case!}\ z\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \vdash_{inf} (D \cup \{z\}, R \cup (R' - \{z\}), S, N)}\ [\text{CASE!}_I]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\} \,.\, R'_{p_i} = \{x \in P_i \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} \\ \forall i \in \{1..n\} \,.\, R''_{p_i} = \{y \in ((D_p \cap N) \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (N_i \cup P_i) \\ \forall i \in \{1..n\} \,.\, (R'_{p_i} \cup R_p) \cap S_i = \emptyset \\ \forall i \in \{1..n\} \,.\, (D_p \cap N_i, (R_p \cup R'_{p_i} \cup R''_{p_i}) - D_i, S_p \cap N_i) \vdash_{check} e_i \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case!}\ z\ \textbf{of}\ \overline{C_i \overline{x_{ij}}^{n_i} \to e_i}^{n}}\ [\text{CASE!}_C]$$

Definimos Γ^P del mismo modo que en el **case** no destructivo:

$$\begin{array}{rcl} \Gamma^P & = & [x : d \mid x \in D \cup D'] \\ & + & [x : r \mid x \in R \cup R'] \\ & + & [x : s \mid x \in S \cup S'] \end{array}$$

En este caso se cumple que $z \in D$, por lo que $\Gamma^P(z) = d$. Definimos Γ^P_0 como el entorno resultante de eliminar el vínculo $[z : d]$ de Γ^P:

$$\Gamma^P_0 = \Gamma^P \upharpoonright (dom(\Gamma^P) - \{z\})$$

En primer lugar demostraremos que $\forall y \in sharerec(z, e')$ se verifica $\forall i \in \{1..n\}. y \notin FV(e_i)$. Esto viene garantizado por el uso del operador $\rhd$ en la regla [CASE!$_I$], que fuerza a que $(R \cup \{z\}) \cap L = \emptyset$.

A continuación se mostrará que $\Gamma'^P_i \vdash e_i : s$, siendo $\Gamma'^P_i = \Gamma^P_0 + [z : r] + \overline{[x_{ij} : t_{ij}]}^{n_i}$. Para ello partimos del hecho de que $\Gamma^P_i \vdash e_i : s$ para cierto Γ^P_i. Bastará demostrar que $\Gamma'^P_i(x) = \Gamma^P_i(x)$ para toda variable $x \in dom(\Gamma^P_i)$:

Sea $x \in dom(\Gamma^P_i)$. Si $x \in P_i$ se distinguen casos:

- $x \in Rec_i$

  Sabemos por el *inh*! de la regla [CASE!$_I$] que $x \notin R_i$, $x \notin S_i$. Si $x \in N_i$ se ha de cumplir que $x \in D'_i$, por el $\vdash_{check}$ realizado en la regla [CASE$_I$].

  Tenemos, por tanto, $x \in D_i \cup D'_i$. La hipótesis de inducción permite establecer $\Gamma^P_i(x) = d$. Asignando $t_{ij} = d$ para que se cumpla $\Gamma^P_i(x) = \Gamma'^P_i(x) = d$ pueden verificarse las condiciones del predicado *inh*! de la regla [CASE!] del sistema de tipos.

- $x \in P_i - Rec_i$

  La demostración es análoga al caso anterior. Esta vez tenemos $x \in S_i \cup S'_i$ y $\Gamma_i^P(x) = \Gamma_i'^P(x) = s$.

En el caso en que $x \notin P_i$ puede seguirse un razonamiento similar al visto para el **case** no destructivo. Del mismo modo, las propiedades (2a), (2b) y (2c) se cumplen trivialmente por la definición de Γ^P. □

6.4. Definiciones recursivas. Punto fijo

El Teorema 1 visto anteriormente excluía las definiciones de funciones recursivas. En esta sección se ampliará el estudio de corrección del algoritmo de inferencia a este tipo de definiciones. El principal problema a tratar será el de la existencia de un punto fijo en las sucesivas iteraciones del algoritmo.

En primer lugar se definirá un orden entre signaturas. Sea $f\ \overline{x_i}^n\ @r = e$ una definición. Una signatura para la misma es una tupla (I_D, I_R, I_S, I_N), donde $I_R = \emptyset$, $I_D \cup I_S \cup I_N = \{1..n\}$, e I_D, I_S e I_N son disjuntos dos a dos.

DEFINICIÓN 5. *Se define el siguiente orden $\sqsubseteq$ entre signaturas:*

$$(I_D, \emptyset, I_S, I_N) \sqsubseteq (I'_D, \emptyset, I'_S, I'_N) \quad \Leftrightarrow_{def} \quad \begin{array}{l} I_D \subseteq I'_D \wedge \\ I_D \cup I_S \subseteq I'_D \cup I'_S \wedge \\ I_N \supseteq I'_N \end{array}$$

Trivialmente se obtiene que $\sqsubseteq$ es reflexivo y transitivo. La propiedad de antisimetría puede deducirse a partir del hecho de que I_D, I_S e I_N son disjuntos dos a dos. Dada una definición $f\ \overline{x_i}^n\ @r = e$, el conjunto de signaturas posibles para la misma junto con el orden $\sqsubseteq$ conforman un retículo finito cuyo elemento mínimo es $\bot = (\emptyset, \emptyset, \emptyset, \{1..n\})$. Este orden puede extenderse a entornos de funciones del siguiente modo. Se cumple $\Sigma \sqsubseteq \Sigma'$ si y sólo si $dom(\Sigma) = dom(\Sigma')$ y para todo símbolo de función $g \in dom(\Sigma) \cap dom(\Sigma')$:

$$\Sigma \vdash g : S \ \wedge\ \Sigma' \vdash g : S' \Rightarrow S \sqsubseteq S'$$

El primer paso consistirá en demostrar que en un recorrido bottom-up del árbol abstracto se obtienen más variables con marca distinta de n cuanto mayor es la signatura de partida. Utilizaremos la notación $e \vdash_{inf}^{\Sigma} (D, R, S, N)$ para indicar que el algoritmo ha inferido los conjuntos (D, R, S, N) para la expresión e en un entorno Σ.

LEMA 5. *Sea $f\ \overline{x_i}^n\ @r = e$ una definición y Σ, Σ' dos entornos de función tales que $\Sigma \sqsubseteq \Sigma'$. Si bajo el entorno Σ el algoritmo obtiene $e \vdash_{inf}^{\Sigma} (D, R, S, N)$ y bajo el entorno Σ' se obtiene $e \vdash_{inf}^{\Sigma'} (D', R', S', N')$, entonces:*

1. $D \cup R \subseteq D' \cup R'$

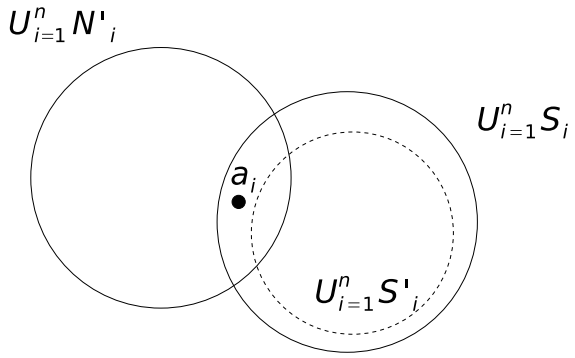

Figura 6.1: Situación en la que $N \subset N'$

2. $D \cup R \cup S \subseteq D' \cup R' \cup S'$

3. $N \supseteq N'$

Demostración. Por inducción estructural sobre e.

$\boxed{e = c}$ $\boxed{e = x}$ $\boxed{e = x@r}$ $\boxed{e = x!}$ $\boxed{e = C\ \overline{a_i}^n\ @r}$

Se cumple trivialmente, ya que los conjuntos (D, R, S, N) obtenidos no dependen del entorno Σ.

$\boxed{e = g\ \overline{a_i}^n\ @r}$

Sea $\Sigma \vdash g : (I_D, \emptyset, I_S, I_N)$ y $\Sigma' \vdash g : (I'_D, \emptyset, I'_S, I'_N)$. A partir de $I_D \subseteq I'_D$ puede obtenerse $D \subseteq D'$ y además $R \subseteq R'$. Por tanto, se tiene $D \cup D' \subseteq R \cup R'$.

Por otro lado, dado que $I_D \cup I_S \subseteq I'_D \cup I'_S$ se cumple $D \cup S \subseteq D' \cup S'$. Por tanto, $D \cup R \cup S \subseteq D' \cup R' \cup S'$.

Demostramos por reducción al absurdo que $N \supseteq N'$. Supongamos que existe un a_i tal que $a_i \notin N$ pero $a_i \in N'$. A partir de la inclusión $I_N \supseteq I'_N$ puede deducirse $\bigcup_{i=1}^{n} N_i \supseteq \bigcup_{i=1}^{n} N'_i$, por lo que la única posibilidad es que $\bigcup_{i=1}^{n} S_i \supset \bigcup_{i=1}^{n} S'_i$. Esta situación se muestra en la Figura 6.1.

Sea $a_i \in \bigcup_{i=1}^{n} N'_i$, esto es, $i \in I'_N$. Si $a_i \in \bigcup_{i=1}^{n} S_i$ pero $a_i \notin \bigcup_{i=1}^{n} S'_i$ se tiene $i \in I_S$, pero $i \notin I'_S$. De esto último puede deducirse $a_i \in I'_D$ por la propiedad $I_D \cup I_S \subseteq I'_D \cup I'_S$. Como I'_D e I'_N son disjuntos se tiene $i \notin I'_N$

$\boxed{e = \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}$

Supongamos que se ha obtenido:

$$e_1 \vdash^{\Sigma}_{inf} (D_1, R_1, S_1, N_1) \quad e_2 \vdash^{\Sigma}_{inf} (D_2, R_2, S_2, N_2)$$
$$e_1 \vdash^{\Sigma'}_{inf} (D'_1, R'_1, S'_1, N'_1) \quad e_2 \vdash^{\Sigma'}_{inf} (D'_2, R'_2, S'_2, N'_2)$$

Podemos deducir (1):

$$\begin{array}{lll} & D \cup R & \\ = & ((D_1 \cup D_2) - \{x_1\}) \cup (R_1 \cup (R_2 - D_1)) & \\ = & (D_1 \cup D_2 \cup R_1 \cup R_2) - \{x_1\} & \text{por } x_1 \notin R_1 \text{ y } x_1 \notin R_2 \\ \subseteq & (D'_1 \cup D'_2 \cup R'_1 \cup R'_2) - \{x_1\} & \text{por H.I.} \\ = & ((D'_1 \cup D'_2) - \{x_1\}) \cup (R'_1 \cup (R'_2 - D'_1)) & \\ = & D' \cup R' & \end{array}$$

De un modo similar puede demostrarse la propiedad (2):

$$\begin{array}{ll} & D \cup R \cup S \\ = & ((D_1 \cup D_2) - \{x_1\}) \cup (R_1 \cup (R_2 - D_1)) \cup (((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2)) \\ = & ((D_1 \cup D_2) \cup (R_1 \cup (R_2 - D_1)) \cup (((S_1 - N_2) \cup S_2) - (D_2 \cup R_2))) - \{x_1\} \\ = & (D_1 \cup D_2 \cup R_1 \cup R_2 \cup (S_1 - N_2) \cup S_2) - \{x_1\} \\ \subseteq & (D'_1 \cup D'_2 \cup R'_1 \cup R'_2 \cup (S'_1 - N'_2) \cup S'_2) - \{x_1\} \\ = & ((D'_1 \cup D'_2) - \{x_1\}) \cup (R'_1 \cup (R'_2 - D'_1)) \cup (((S'_1 - N'_2) \cup S'_2) - (\{x_1\} \cup D'_2 \cup R'_2)) \\ = & D' \cup R' \cup S' \end{array}$$

Con respecto a la propiedad (3) sólo es necesario utilizar la hipótesis de inducción y la propiedad (2).

$$\boxed{e = \textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

Supongamos que se obtiene para todo $i \in \{1..n\}$:

$$e_i \vdash^{\Sigma}_{inf} (D_i, R_i, S_i, N_i) \qquad e_i \vdash^{\Sigma'}_{inf} (D'_i, R'_i, S'_i, N'_i)$$

Aplicando la hipótesis de inducción puede obtenerse fácilmente:

$$D \cup R = \bigcup_{i=1}^{n} (D_i - P_i) \cup \bigcup_{i=1}^{n} (R_i - P_i) \subseteq \bigcup_{i=1}^{n} (D'_i - P_i) \cup \bigcup_{i=1}^{n} (R'_i - P_i) = D' \cup R'$$

Con lo que queda demostrada la propiedad (1). Para el resto de propiedades el razonamiento es similar.

$$\boxed{e = \textbf{case!} \ x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}$$

La demostración es análoga a la del **case** no destructivo. El conjunto R' que aparece en la regla [CASE!$_I$] no afecta en la demostración, ya que es independiente del entorno utilizado. □

DEFINICIÓN 6. *Para cada declaración $f\ \overline{x_i}^n\ @r = e$ se define la función F definida sobre entornos de la siguiente forma:*

$$F_{f\ \overline{x_i}^n\ @r=e}(\Sigma) = \Sigma\,[g \mapsto extract(\overline{x_i}^n, D, R, S, N)] \ \textbf{\textit{donde}}\ e \vdash^{\Sigma}_{inf} (D, R, S, N)$$

donde la función extract se encarga de obtener la signatura de la función a partir de los conjuntos (D, R, S, N) correspondientes y está definida del siguiente modo:

$$\begin{aligned} extract(\overline{x_i}^n, D, \emptyset, S, N) \;=\; & (\{i \in \{1..n\} \mid x_i \in D\}, \\ & \emptyset, \\ & \{i \in \{1..n\} \mid x_i \in S\}, \\ & \{i \in \{1..n\} \mid x_i \in N\} \cup \{i \in \{1..n\} \mid x_i \notin D \cup S \cup N\}) \end{aligned}$$

LEMA 6. *Dada una definición $f\ \overline{x_i}^n\ @r = e$, la función $F_{f\ \overline{x_i}^n\ @r=e}$ es monótona con el orden $\sqsubseteq$. Es decir:*

$$\Sigma \sqsubseteq \Sigma' \Longrightarrow F_{f\ \overline{x_i}^n\ @r=e}(\Sigma) \sqsubseteq F_{f\ \overline{x_i}^n\ @r=e}(\Sigma')$$

Demostración. Sean los siguientes resultados obtenidos por el algoritmo de inferencia:

$$\begin{aligned} e \vdash^{\Sigma}_{inf} (D, R, S, N) \qquad & (I_D, \emptyset, I_S, I_N) = extract(\overline{x_i}^n, D, R, S, N) \\ e \vdash^{\Sigma'}_{inf} (D', R', S', N') \qquad & (I'_D, \emptyset, I'_S, I'_N) = extract(\overline{x_i}^n, D', R', S', N') \end{aligned}$$

Demostramos que $(I_D, \emptyset, I_S, I_N) \sqsubseteq (I'_D, \emptyset, I'_S, I'_N)$. Por el Lema 5 se cumple $D \cup R \subseteq D' \cup R'$. Dado que $R = R' = \emptyset$, tenemos $D \subseteq D'$ y por tanto, $I_D \subseteq I'_D$. El resto de condiciones se obtienen directamente a partir de las propiedades enunciadas en el Lema 5. □

Este lema establece que en cada iteración del algoritmo se obtiene una signatura cada vez mayor y que por tanto, puede alcanzarse el punto fijo mediante la cadena ascendente de Kleene. Esto nos permitirá modificar el Teorema 1 para incluir definiciones recursivas:

TEOREMA 2. *Supongamos que la declaración de función $f\ \overline{x_i}^n = e$ ha sido aceptada por el algoritmo de inferencia (incl. cálculo de punto fijo y* check *final) y sea e' cualquier subexpresión de e para la cual el algoritmo ha obtenido $e' \vdash_{inf} (D, R, S, N)$ y $(D', R', S') \vdash^*_{check} e'$. Entonces existe un tipo seguro s y un entorno Γ^P que cumplen:*

1. $\Gamma^P \vdash e' : s$
2. $\forall x \in scope(e')$:
   a) $\Gamma^P(x) = d \Leftrightarrow x \in D \cup D'$
   b) $\Gamma^P(x) = s \Leftrightarrow x \in S \cup S'$
   c) $\Gamma^P(x) = r \Leftrightarrow x \in R \cup R'$

Demostración. La demostración es similar a la del Teorema 1. El único caso adicional que hay considerar es aquel en que se realiza una llamada recursiva a la función que se está definiendo, es decir, $e' = f\ \overline{x_i}^n\ @r$. Sea $(I_D, \emptyset, I_S, I_N)$ la signatura obtenida para f. Esta vez I_N podría ser no vacío. Para cada $i \in \{1..n\}$ puede definirse Γ^P_i de esta forma:

$$\Gamma^P_i = \begin{cases} [a_i : d] & \text{si } a_i \text{ es variable y } a_i \in D \cup D' \\ [a_i : s] & \text{si } a_i \text{ es variable y } a_i \in S \cup S' \end{cases}$$

Por su parte, definimos Γ'^P del mismo modo que en la aplicación de función no recursiva:

$$\Gamma'^P = [r : \rho, g : \sigma] + \bigoplus_{i=0}^{n} \Gamma_i^P$$

Se mostrará que Γ'^P está bien definido. En efecto, supongamos que $x \in dom(\Gamma_i^P) \cap dom(\Gamma_j^P)$ tal que $\Gamma_i^P(x) = \Gamma_j^P(x) = d$ para ciertos $i, j \in \{1..n\}$ tales que $i \neq j$. Si $x \in D$ puede aplicarse el mismo razonamiento visto para el Teorema 1 para así obtener una contradicción. Tampoco puede obtenerse $x \in D'$ por la siguiente condición de la regla [APP$_C$], que prohíbe utilizar un mismo parámetro en dos posiciones destructivas:

$$\forall a_i \in D_p.\, (\#j : 1 \leq j \leq n : a_i = a_j) = 1$$

El resto de la demostración es similar a la vista para el Teorema 1 en el caso de funciones no recursivas. □

Capítulo 7

Conclusiones y trabajo futuro

En este trabajo se ha presentado un algoritmo de inferencia en el contexto de SAFE, un lenguaje funcional con liberación explícita de memoria. Dicho algoritmo permite asegurar que un programa en ejecución se comportará conforme a una de las normas de la política de seguridad de SAFE: la ausencia de accesos a partes de memoria ya liberadas.

7.1. Trabajo relacionado

7.1.1. Manejo de memoria mediante regiones

El uso de regiones en los lenguajes funcionales con el fin de evitar la necesidad de un recolector de basura no es una novedad en este lenguaje. Se han propuesto varios enfoques destinados a tratar este problema. En [HMN01] se realiza una comparación de algunos de ellos, usando un ejemplo que implementa el juego de la vida:

$$
\begin{array}{l}
\mathit{nextgen}\ g = \{\text{calcular nueva generación y devolverla}\} \\
\mathit{life}\ n\ g = \textbf{if}\ n = 0\ \textbf{then}\ g \\
\qquad\qquad \textbf{else}\ \mathit{life}\ (n-1)\ (\mathit{nextgen}\ g)
\end{array}
$$

Supongamos que una generación g es una estructura de datos de gran tamaño situada en la memoria. Una función como ésta reservaría n generaciones en memoria hasta que un recolector de basura decidiese liberar las generaciones intermedias. Sin embargo, si el único uso que se le da a una generación intermedia es la creación de la generación siguiente, sería razonable liberar la estructura de datos intermedia tan pronto como fuera posible. En SAFE puede modificarse el programa del siguiente modo para obtener tal comportamiento:

$$
\begin{array}{l}
\mathit{nextgen}\ g\ r = \textbf{case!}\ g\ \textbf{of} \rightarrow \ldots \\
\qquad\qquad \{\text{calcular nueva generación en la región } r\} \\
\mathit{life}\ n\ g\ r = \textbf{if}\ n = 0\ \textbf{then}\ g! \qquad \{\text{reutilizar argumento } g\} \\
\qquad\qquad \textbf{else}\ \mathit{life}\ (n-1)\ (\mathit{nextgen}\ g@r)@r
\end{array}
$$

Tofte y Talpin [TT97] introducieron el uso de regiones anidadas mediante la extensión de Core ML con una construcción **letregion** ρ. Al igual que en SAFE, las regiones son áreas de memoria donde se construyen estructuras de datos, y se reservan y liberan en su totalidad. Una diferencia con el sistema de gestión de memoria en SAFE es que en nuestro lenguaje la reserva y liberación de memoria están sincronizadas con las llamadas a función. Más importante resulta el hecho de que en SAFE se permite la destrucción selectiva de estructuras de datos en la región de trabajo y la región de salida. En la propuesta original de [TT97] se obligaría que todas las generaciones del juego de la vida convivieran en una misma región, lo cual no permitiría obtener ninguna ventaja en el uso de la memoria. Sin embargo, una extensión a este trabajo [BTV96, TBE+06] permite *reiniciar* todas las estructuras de datos contenidas en una región sin necesidad de liberar la región completa. Esto permite, en el ejemplo anterior, que la generación anterior se reinicie una vez creada la nueva generación. De este modo se necesitaría una nueva región temporal para reservar la generación de nueva, que será copiada en la región de salida tras reiniciar esta última. El programador no necesita anotar la funcion con ningún tipo de información que especifique el reinicio de una región, pero sí ha de introducir las funciones de copia correspondientes.

$$
\begin{array}{l}
\mathit{nextgen}\ g = \{\text{calcular nueva generación y devolverla}\} \\
\mathit{life}\ n\ g = \textbf{if}\ n = 0\ \textbf{then}\ g \\
\qquad\qquad \textbf{else}\ \mathit{life}\ (n-1)\ (\textbf{copy}\ (\mathit{nextgen}\ g))
\end{array}
$$

La función *copy* permite construir ahora la nueva generación en una región separada, con lo que es posible ejecutar la función *life* con un coste en espacio constante. No obstante, esta versión requiere realizar una copia completa de una generación en cada llamada recursiva, lo cual aumenta el tiempo de ejecución. Además la inserción de la función *copy* requiere un profundo conocimiento del mecanismo de reinicio, ya que éste no aparece de forma explícita en el programa.

El sistema AFL [AFL95] inserta órdenes de reserva y liberación de memoria de forma independiente a la construcción **letregion**, que en este momento sólo introduce nuevas regiones en ámbito. En el ejemplo que se está tratando, esto permite liberar la región antigua tan pronto como se calcula la generación nueva, sin necesidad de ninguna copia en la llamada recursiva. Tan sólo se necesita una copia en el caso base:

$$
\begin{array}{l}
\mathit{nextgen}\ g = \{\text{calcular nueva generación y devolverla}\} \\
\mathit{life}\ n\ g = \textbf{if}\ n = 0\ \textbf{then copy}\ g \\
\qquad\qquad \textbf{else}\ \mathit{life}\ (n-1)\ (\mathit{nextgen}\ g)
\end{array}
$$

De nuevo, la inserción de la función *copy* por parte del programador requiere un profundo conocimiento de las anotaciones incorporadas por el compilador tras el análisis.

Como ventaja de estos enfoques de manejo de memoria sobre el enfoque adoptado por SAFE podemos destacar la ausencia de anotaciones de región en los primeros. Desde el punto de vista del programador resulta engorroso incluir anotaciones *@r* en

cada construcción y en cada llamada a función que construya una estructura de datos que devuelva como resultado. Actualmente estamos trabajando en un algoritmo de inferencia de regiones que permita al compilador incorporar estas anotaciones automáticamente.

Hughes y Pareto [HP99b] también incorporan el concepto de región en su sistema de tipos con tamaños. De este modo, el consumo de memoria de la pila y el montón puede ser comprobado mediante este sistema de tipos. En este enfoque los tamaños de región están acotados. De nuevo, la diferencias de este sistema con la gestión de memoria en SAFE es la sincronización de llamadas a funciones con creación de regiones y la destrucción explícita de estructuras de datos. No obstante, SAFE no proporciona de momento ningún mecanismo para inferir el tamaño de región necesario para ejecutar un programa (ver trabajo futuro).

Con respecto a otros trabajos que involucren destrucción explícita en programación funcional, Hofmann y Jost [HJ03] desarrollaron un sistema de tipos para inferir consumo de memoria. En este trabajo proponen un lenguaje funcional con una construcción *match*, en el cual está inspirada la construcción **case**!. El sistema de Hofmann y Jost permite asociar una función con una cota superior del espacio necesario para su ejecución y el espacio de memoria restante tras la misma. Sin embargo, su sistema de tipos no proporciona en tiempo de compilación la seguridad de estas funciones de destrucción.

7.1.2. Tipos lineales

El sistema de tipos de SAFE posee algunas características comunes con los tipos lineales desarollados por Wadler (una referencia básica puede encontrarse en [Wad90]). Los valores que tengan un tipo lineal no pueden ser nombrados en el texto más que una vez. Este tipo de valores no requieren ningún contador de referencias ni recolección de basura. En el marco del sistema de tipos SAFE, las variables con tipo condenado no pueden ser nombradas ni destruidas tras la destrucción, mientras que las variables con tipo seguro no tienen ninguna limitación con respecto al número de accesos realizados. En este sentido, el sistema de tipos SAFE es lineal con respecto a la destrucción, pero no con respecto a la lectura.

7.2. Conclusiones

En este trabajo se ha descrito el diseño e implementación de tres fases fundamentales en el compilador de SAFE: la inferencia de tipos Hindley-Milner, el análisis de compartición y el análisis de tipos SAFE.

Una vez implementada, se ha aplicado la herramienta a pequeños casos de estudio y se ha podido comprobar que en muchas funciones elementales que hacen uso de las facilidades de destrucción son aceptadas por el algoritmo.

Por otra parte, en el caso de funciones no elementales se ha detectado la principal causa por la que algunas de ellas son rechazadas, lo cual nos ha permitido identificar

los puntos a mejorar del sistema.

7.3. Trabajo futuro

El comportamiento del algoritmo de inferencia para el ejemplo de *Mergesort* revela la necesidad de un análisis de compartición más preciso que el descrito en este capítulo. Como ya fue explicado en la Sección 5.6 la debilidad de este análisis radica en el caso de la aplicación de función. Actualmente estamos estudiando posibles mejoras en la definición de la signatura de compartición de una función. Además de distinguir entre relaciones de compartición entre estructuras recursivas y no recursivas, es necesario incorporar información adicional que reduzca el número de posibles candidatos a aparecer en el resultado de la interpretación S para una llamada a una función.

Otro trabajo ya en curso pero actualmente no implementado en el compilador es un mecanismo de inferencia de regiones para programas *Full-Safe*. Mediante este mecanismo el compilador insertará automáticamente la región de destino en expresiones de copia, aplicación de función y aplicación de constructor, de modo que el programador queda liberado de la tarea de insertar manualmente dichas anotaciones. Parte de este mecanismo de inferencia consiste en introducir variables de región arbitrarias en las expresiones que lo requieran y realizar un análisis para inferir cuales de ellas pueden asimilarse a la región de salida y a la región de los parámetros de entrada.

Entre los objetivos a largo plazo del proyecto está la inferencia de terminación para programas SAFE y la inferencia de una cota superior de consumo de memoria. El primer objetivo es un trabajo en curso [LP07] que hace uso de técnicas de terminación de sistema de reescritura basados en interpretaciones polinómicas. Con respecto al consumo de memoria de un programa, este objetivo está relacionado con el primero y permite satisfacer una de las normas que conforman la política de seguridad de programas SAFE.

Ya en el marco del código con demostración asociada y una vez realizados estos análisis, podrá incluirse una demostración (certificado) en el código objeto generado, de modo que el consumidor de dicho código pueda comprobar que satisface las propiedades de destrucción segura, terminación y cota de uso de memoria mediante el uso de un verificador independiente. Con respecto a este último puede utilizarse algún asistente de demostraciones, como Coq o Isabelle.

Bibliografía

[Ada93] S. Adams. Efficient sets –a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[AFL95] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, PLDI'95*, pages 174–185. ACM Press, 1995.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

[CB83] J. Corbin and M. Bidoit. A rehabilitation of robinson's unification algorithm. In R.E.A. Mason, editor, *Proceedings of the 9th World Computer Congress, IFIP'83*, pages 909–914, 1983.

[CLL06] J. Conesa, R. López, and A. Lozano. Desarrollo de un compilador para un lenguaje funcional con gestión explícita de la memoria, Junio 2006.

[DJM05] C. Dornan, I. Jones, and S. Marlow. *Alex user guide*, 2005.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197. ACM Press, 2003.

[HMN01] F. Henglein, H. Makholm, and H.Ñiss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 175–186. ACM Press, 2001.

[HP99a] R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.

[HP99b] R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.

[Jon87] S.L. Peyton Jones. *The Implementation of Funcional Programming Languages*. Prentice Hall, 1987.

[LP07] S. Lucas and R. Peña. Termination and complexity bounds for safe programs. In E. Pimentel, editor, *VII Jornadas sobre Programación y Lenguajes (PROLE '07)*, pages 233–242. Thomson, 2007.

[MG01] S. Marlow and Andy Gill. *Happy user guide*, 2001.

[MPS07] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'06*, pages XVI1 – XVI16, 2007.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS04] R. Peña and C. Segura. A first-order functional language for reasoning about heap consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04. Technical Report 0408, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel*, pages 64–80, 2004.

[PSM06] R. Peña, C. Segura, and M. Montenegro. A sharing analysis for safe. In *Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP'06*, pages 205–221, 2006.

[PSM07] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Trends in Functional Programming (Volume 7) Selected Papers of the Seventh Symposium on Trends in Functional Programming, TFP'06.* Intellect, 2007.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[TBE+06] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, 2006.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.